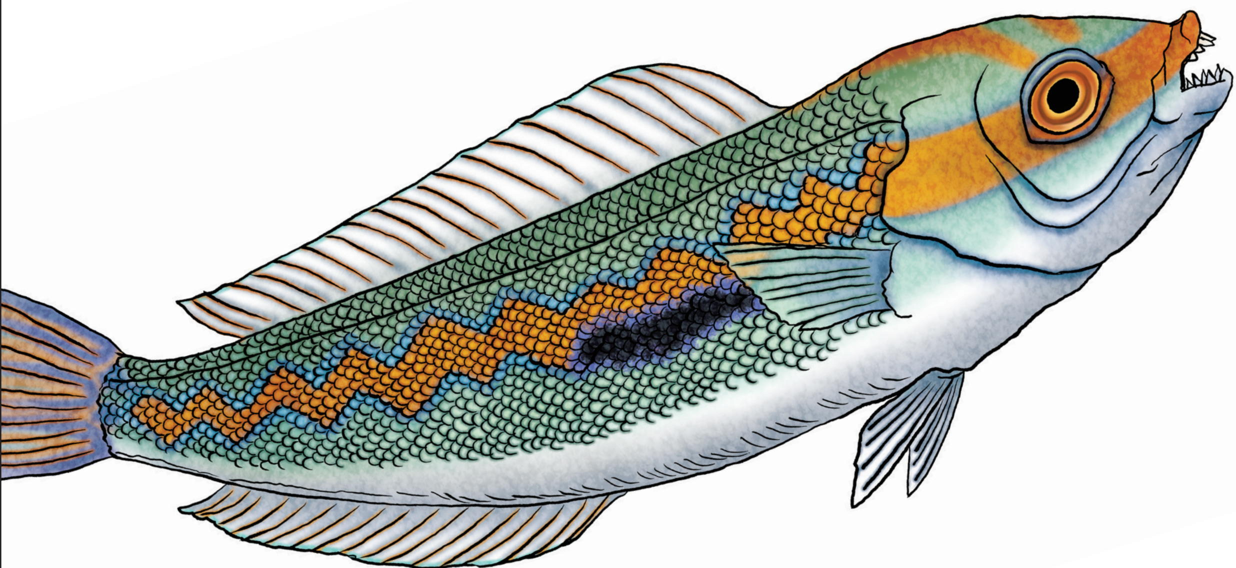
O'REILLY®

# The Practitioner's Guide to Graph Data

## Applying Graph Thinking and Graph Technologies to Solve Complex Problems



Denise Koessler Gosnell &
Matthias Broecheler

O'REILLY®

# The Practitioner's Guide to Graph Data

Graph data closes the gap between the way humans and computers view the world. While computers rely on static rows and columns of data, people navigate and reason about life through relationships. This practical guide demonstrates how graph data brings these two approaches together. By working with concepts from graph theory, database schema, distributed systems, and data analysis, you'll arrive at a unique intersection known as graph thinking.

Authors Denise Koessler Gosnell and Matthias Broecheler show data engineers, data scientists, and data analysts how to solve complex problems with graph databases. You'll explore templates for building with graph technology, along with examples that demonstrate how teams think about graph data within an application.

- **Build an example application architecture with relational and graph technologies**
- **Use graph technology to build a Customer 360 application, the most popular graph data pattern today**
- **Dive into hierarchical data and troubleshoot a new paradigm that comes from working with graph data**
- **Find paths in graph data and learn why your trust in different paths motivates and informs your preferences**
- **Use collaborative filtering to design a Netflix-inspired recommendation system**

**Denise Koessler Gosnell, PhD,** is chief data officer of DataStax. She's built and patented dozens of processes related to graph theory, graph algorithms, and graph data applications.

**Matthias Broecheler, PhD,** is chief technologist at DataStax. A graph database expert, he invented the Titan graph database.

"A must-have addition to any coder's arsenal of references. Both authors are exemplars in the field of graph theory, architecture, and principles."

**—Theodore C. Tanner Jr.**
Global Chief Technology Officer and Chief Architect, Watson Health

"This book does a masterful job of leveling up your graph database knowledge. It's the perfect starting point for newcomers, and even the most seasoned practitioners will learn new things."

**—Matthew Russell**
Chief Executive Officer, Strongest AI, and author of *Mining the Social Web*

AI / SEMANTICS

US $69.99 CAN $92.99
ISBN: 978-1-492-04407-9

Twitter: @oreillymedia
facebook.com/oreilly

# The Practitioner's Guide to Graph Data

*Denise Koessler Gosnell and Matthias Broecheler*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**The Practitioner's Guide to Graph Data**
by Denise Koessler Gosnell and Matthias Broecheler



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Acquisitions Editor:** Jonathan Hassell
**Developmental Editor:** Jeff Bleiel
**Production Editor:** Nan Barber
**Copyeditor:** Arthur Johnson
**Proofreader:** Josh Olejarz
**Indexer:** Ellen Troutman-Zaig
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

April 2020: First Edition

**Revision History for the First Edition**
2020-03-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492044079* for release details.



This work is part of a collaboration between O'Reilly and DataStax. See our statement of editorial independence.

978-1-492-04407-9

[LSI]

# Table of Contents

# Preface

Think about the last time you searched for someone on a social media platform.

What did you look at on the results page?

Most likely, you started scanning down the names in the list of profile results. And you probably spent most of your time inspecting the "shared friends" section to understand how you knew someone.

Our innate human behavior of reasoning about our shared friends on social media is what inspired us to write this book. Though, our shared inspiration generated two very different reasons behind why we wrote this book.

First, have you ever stopped to think about *how* an app creates the "shared friends" section?

The engineering required to deliver your "shared friends" in search results creates an intricate orchestration of tools and data to solve an extremely complex, distributed problem. We have either built those sections or created the tools that deliver them. Our passion for understanding and teaching others from our collective experiences is the first reason we chose to write this book together.

The second reason is that anyone who uses social media intuitively derives personal context directly from the "shared friends" section. This process of reasoning and thinking about relationships within data is called *graph thinking*, and that is what we name the human approach to understanding life through connected data.

How did we all learn to do this?

There wasn't a specific point in time when we all were taught this skill. Processing relationships among people, places, or things is just how we think.

It is the ease with which people infer context from relationships, be it in real life or from data, that has ignited the wave of graph thinking.

And when it comes to understanding graph thinking, most people fall into one of two camps: those who think graphs are about bar charts, or those who think graphs are way too complicated. Either way, these thought processes apply legacy approaches to thinking about data and technology. The problem is that the art of the possible has changed, our tools have improved, and there are new lessons to learn.

We believe that graphs are powerful and deployable. Graph technology can make you more productive; we have worked with teams that told us so.

This book brings these two mindsets together.

Graph thinking closes the gap between how we humans operate/see/live and how we use data to inform a decision. Imagine seeing your whole world as a spreadsheet with rows and columns of data and trying to make sense of it all. For the majority of us, the exercise is unnatural and counterproductive.

This is because relationships are how people navigate and reason about life. It is computers that need databases and operate in the world of rows and columns of data.

Graph thinking is a way to solve complex problems by taking a relationship-centric approach. Graph technology bridges the gap between "relationships" and the linear memory constraints of modern computer infrastructure.

As more people learn how to build with graph technology by applying graph thinking, imagine what the next wave of innovation will bring.

## Who Should Read This Book

This book aims to teach you two things. First, we will teach you about graph thinking and the graph mindset through asking questions and reasoning about data. Second, we will walk you through writing code that solves the most common, complex graph problems.

These new concepts are intertwined within the tasks commonly performed across a few different engineering functions.

Data engineers and architects sit at the heart of transitioning an idea from development into production. We organized this book to show you how to resolve common assumptions that can occur when moving from development into production with graph data and graph tools. Another benefit to the data engineer or data architect will be learning the world of possibilities that come from understanding graph thinking. Synthesizing the breadth of problems that can be solved with graph data will also help you invent new patterns for their use in production applications.

Data scientists and data analysts may most benefit from reasoning about how to use graph data to answer interesting questions. All the examples throughout this text were constructed to apply a query-first approach to graph data. A secondary benefit

for a data scientist or analyst will be to understand the complexity of using distributed graph data within a production application. We teach and build upon the common development pitfalls and their production resolution processes throughout the book so that you can formulate new types of problems to solve.

Computer scientists will learn how to use techniques in functional programming and distributed systems to query and reason about graph data. We will outline fundamental approaches to procedurally traversing graph data and step through their application with graph tools. Along the way we will learn about distributed technologies, too.

We will be working within the intersection of graph data and distributed, complex problems; a fascinating combination of engineering topics with something to learn for any technologist.

## Goals of This Book

The first goal of this book is to create a new foundation that exists at a very diverse intersection. We will be working with concepts from graph theory, database schema, distributed systems, data analysis, and many other fields. This unique intersection forms what we refer to in this book as *graph thinking*. A new application domain requires new terms, examples, and techniques. This book serves as your foundation for understanding this emerging field.

From the past decade of graph technology emerged a common set of patterns for using graph data in production applications. The second goal of this book is to teach you those patterns. We define, illustrate, build, and implement the most popular ways teams use graph technology to solve complex problems. After studying this book, you will have a set of templates for building with graph technology to solve this common set of problems.

The third goal of this book is to transform how you think. Understanding and applying graph data to your problem introduces a paradigm shift into your thought processes. Through many upcoming examples, we aim to teach you the common ways that others think and reason about graph data within an application. This book teaches you what you need to know to apply graph thinking to a technology decision.

## Navigating This Book

This book is organized roughly as follows:

- Chapter 1 discusses graph thinking and provides detailed processes for its application to complex problems.
- Chapters 2 and 3 introduce fundamental graph concepts that will be used throughout the rest of the book.

- Chapters 4 and 5 apply graph thinking and distributed graph technology to building a Customer 360 banking application, the most popular use case for graph data today.
- Chapters 6 and 7 into the world of hierarchical data and nested graph data through a telecommunications use case. Chapter 6 sets the stage for a common error that is resolved in Chapter 7.
- Chapters 8 and 9 discuss pathfinding across graph data in detail, using an example of quantifying trust in social transaction networks via paths.
- Chapters 10 and 12 teach you how to use collaborative filtering on graph data to design a Netflix-inspired recommendation system.
- Chapter 11 can be thought of as a bonus chapter that illustrates how to apply entity resolution to the merging of multiple datasets into one large graph for collective analysis.

Each chapter pair (4 and 5, 6 and 7, 8 and 9, 10 and 12) follows the same structure. The first chapter in each pair introduces new concepts and a new example use case in a development environment. The second chapter delves into the details of production issues, such as performance and scalability, that need to be addressed for real-world deployments.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/datastax/graph-book*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

You can also follow us on Twitter: *https://twitter.com/Graph_Thinking*

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*The Practitioner's Guide to Graph Data* by Denise Koessler Gosnell and Matthias Broecheler (O'Reilly). Copyright 2020 Denise Gosnell and Matthias Broecheler, 978-1-492-04407-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

## O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

# How to Contact Us

Please address comments and questions concerning this book to the publisher:


O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)


We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781492044079*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

# Acknowledgments


We would like to thank the incredible group of people who donated their time and expertise to advising us and to reading, and correcting this book.

We had the honor of working with an world-class editing team led by Jeff Bleiel. Our technical editing team of Alexey Ott, Lorina Poland, and Daniel Kuppitz applied their seasoned experience in creating, building, and writing about graph technologies. Their direct contributions elevated this book to a level that we could have reached only with their assistance. We are humbled that they went above and beyond to improve the quality and correctness of this text. Thank you.

We also would like to thank DataStax for its sponsorship and for encouraging our teams to collaborate on creating this book. We are very grateful for the support and review by the DataStax Graph Engineering team and for the product changes they made as we created our work together: Eduard Tudenhoefner, Dan LaRocque, Justin Chu, Rocco Varela, Ulises Cerviño Beresi, Stephen Mallette, and Jeremiah Jordan. We are especially grateful to Bryn Cooke, who coordinated and implemented a nontrivial amount of extra work to support the ideas in this book.

Many additional people transcended their obligations to make time to support us, as is the DataStax way. We would like to thank Dave Bechberger, Jonathan Lacefield, and Jonathan Ellis for their expert contributions and advocacy for this work. To Daniel Farrell, Jeremy Hanna, Kiyu Gabriel, Jeff Carpenter, Patrick McFadin, Peyton Casper, Matt Atwater, Paras Mehra, Kelly Mondor, and Jim Hatcher: our conversations throughout the creation of this work had more of an impact than you realize, so thank you.

All of the stories and examples throughout this text were inspired by our collaborations and experiences with colleagues around the world. To that end, we would like to recognize the graph heroes who spoke with us and helped shape this book's narrative: Matt Aldridge, Christine Antonsen, David Boggess, Sean Brandt, Vamsi Duvvuri, Ilia Epifanov, Amy Hodler, Adam Judelson, Joe Koessler, Eric Koester, Alec Macrae, Patrick Planchamp, Gary Richardson, Kristin Stone, Samantha Tracht, Laurent Weichberger, and Brent Woosley. The time that we spent speaking with each of you and the information you shared made its way into the stories that we have the privilege of sharing in this text. Thank you for lending your voices, experiences, and ideas.

Denise would also like to extend her personal gratitude to those who mentored her throughout this journey. To Teresa Haynes and Debra Knisley: you ignited my passion for graph theory that continues to drive me every day; I wouldn't have started this journey without you. To Mike Berry: you taught me how to get things done and to never stop reaching for my next big idea; thank you. To Ted Tanner: you opened a door and showed me what it means to build with passion and deliver with excellence; timing and execution are everything. To Mike Canzoneri: whether you know it or not, you were the boot that kicked me over the line to write this; thank you. And most importantly, to Ty, the unofficial "third author" who was with me every step of the way: thank you for your never-ending positivity.

CHAPTER 1

# Graph Thinking

Think about the first time you learned about graph technology.

The scene probably started at the whiteboard where your team of directors, architects, scientists, and engineers were discussing your data problems. Eventually, someone drew the connections from one piece of data to another. After stepping back, someone noted that the links across the data built up a graph.

That realization sparked the beginning of your team's graph journey. The group saw that you could use relationships across the data to provide new and powerful insights to the business. An individual or a small group was probably tasked with evaluating the techniques and tools available for storing, analyzing, and/or retrieving graph-shaped data.

The next major revelation for your team was likely that it's easy to explain your data as a graph. But it's hard to use your data as a graph.

Sound familiar?

Much like this whiteboard experience, earlier teams discovered connections within their data and turned them into valuable applications we use everyday. Think about apps like Netflix, LinkedIn, and GitHub. These products translate connected data into an integral asset used by millions of people around the world.

We wrote this book to teach you how they did it.

As both tool builders and tool users, we have had the opportunity to sit on both sides of the whiteboard conversation hundreds of times. From our experiences, we collected a core set of choices and subsequent technology decisions to accelerate your journey with graph technology.

This book will be your guide in navigating the space between understanding your data as a graph and using your data as a graph.

## Why Now? Putting Database Technologies in Context

Graphs have been around for centuries. So why are they relevant *now*?

And before you skip this section, we ask you to hear us out. We are about to go into history here; it isn't long, and it isn't involved. We need to do this because the successes and failures of our recent history explain why graph technology is relevant again.

Graphs are relevant now because the tech industry's focus has shifted over the last few decades. Previously, technologies and databases focused on how to *most efficiently* store data. Relational technologies evolved as the front-runner to achieve this efficiency. Now we want to know how we can get the *most value* out of data.

Today's realization is that data is inherently more valuable when it is connected.

A little bit of historical context on the evolution of database technologies sheds a lot of light on how we got here, and maybe even on why you picked up this book. The history of database technology can loosely be divided into three eras: hierarchical, relational, and NoSQL. The following abbreviated tour explores each of these historical eras, with a focus on how each era is relevant to this book.

> The following sections provide you with an abridged version of the evolution of graph technology. We are highlighting only the most relevant parts of our industry's vast history. At the very least, we are saving you from losing your valuable time down the rabbit hole of a self-guided Wikipedia link walking tour—though ironically, the self-guided version would be walking through today's most accessible knowledge graph.

This brief history will take us from the 1960s to today. Our tour will culminate with the fourth era of graph thinking that is on our doorstep, as shown in Figure 1-1. We are asking you to take this short journey with us because we believe that historical context is one of the keys to unlocking the wide adoption of graph technologies within our industry.

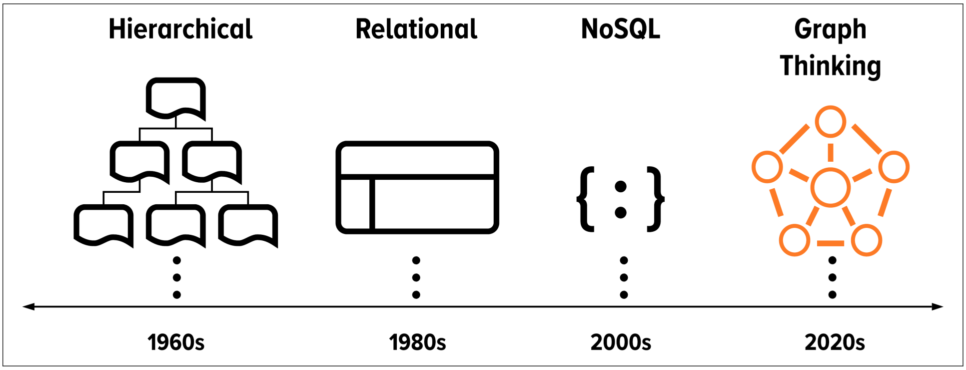



*Figure 1-1. A high-level timeline showing the historical context on the evolution of database technology to illustrate the emergence of graph thinking*

## 1960s–1980s: Hierarchical Data

Technical literature interchangeably labels the database technologies of the 1960s through the 1980s as "hierarchical" or "navigational." Irrespective of the label, the thinking during this era aimed to organize data in treelike structures.

During this era, database technologies stored data as records that were linked to one another. The architects of these systems envisioned walking through these treelike structures so that any record could be accessed by a key or system scan or through navigating the tree's links.

In the early 1960s, the Database Task Group within CODASYL, the Conference/Committee on Data Systems Languages, organized to create the industry's first set of standards. The Database Task Group created a standard for retrieving records from these tree structures. This early standard is known as "the CODASYL approach" and set the following three objectives for retrieving records from database management systems:[1]

1. Using a primary key
2. Scanning all the records in a sequential order
3. Navigating links from one record to another

1 T. William Olle, *The CODASYL Approach to Data Base Management* (Chichester, England: Wiley-Interscience, 1978). No. 04; QA76. 9. D3, O5.

CODASYL was a consortium formed in 1959 and was the group responsible for the creation and standardization of COBOL.

Aside from the history lesson, there is an ironic point we are building up to. At the inception of this approach, the technologists of CODASYL envisioned retrieving data by keys, scans, and links. To date, we have seen significant innovation in and adoption of two of these three original standards: keys and scans.

But what happened with the third goal of CODASYL's retrieval standardization: to navigate links from one record to another? Storing, navigating, and retrieving records according to the links between them describes what we refer to today as graph technology. And as we mentioned before, graphs are not new; technologists have been using them for years.

The short version of this part of our history is that CODASYL's link-navigating technologies were too difficult and too slow. The most innovative solutions at the time introduced B-trees, or self-balancing tree data structures, as a structural optimization to address performance issues. In this context, B-trees helped speed up record retrieval by providing alternate access paths across the linked records.[2]

Ultimately, the imbalance among implementation expenditures, hardware maturity, and delivered value resulted in these systems being shelved for their speedier cousin: relational systems. As a result, CODASYL no longer exists today, though some of the CODASYL committees continue their work.

## 1980s–2000s: Entity-Relationship

Edgar F. Codd's idea to separate the organization of data from its retrieval system ignited the next wave of innovation in data management technologies.[3] Codd's work founded what we still refer to as the entity-relationship era of databases.

The entity-relationship era encompasses the decades when our industry polished the approach for retrieving data by a key, which was one of the objectives set by the early working groups of the 1960s. During this era, our industry developed technology that was, and still is, extremely efficient at storing, managing, and retrieving data from tables. The techniques developed during these decades are still thriving today because they are tested, documented, and well understood.

2 Rudolph Bayer and Edward McCreight, "Organization and Maintenance of Large Ordered Indexes," in *Software Pioneers*, ed. Manfred Broy and Ernst Denert (Berlin: Springer-Verlag, 2002), 245–262.

3 Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377-387.

The systems of this era introduced and popularized a specific way of thinking about data. First and foremost, relational systems are built on the sound mathematical theory of relational algebra. Specifically, relational systems organize your data into sets. These sets focus on the storage and retrieval of real-world entities, such as people, places, and things. Similar entities, such as people, are grouped together in a table. In these tables, each record is a row. An individual record is accessed from the table by its primary key.

In relational systems, entities can be linked together. To create links between entities, you create more tables. A linking table will combine the primary keys of each entity and store them as a new row in the linking table. This era, and the innovators within it, created the solution for tabular-shaped data that still thrives today.

There are volumes of books and more resources than one can mention on the topic of relational systems. This book does not intend to be one of them. Instead, we want to focus on the thought processes and design principles that have become widely accepted today.

For better or for worse, this era introduced and ingrained the mentality that all data maps to a table.

If your data needs to be organized in and retrieved from a table, relational technologies remain the preferred solution. But however integral their role remains, relational technologies are not a one-size-fits-all solution.

The late '90s brought early signs of the information age through the popularization of the web. This stage during our short history hinted at volumes and shapes of data that were previously unplanned and unused. At this time in database innovation, incomprehensible volumes of data in diverse shapes began to fill the queues of applications. A key realization at this point was that the relational model was lacking: there was no mention of the intended use for the data. The industry had a detailed storage model, but nothing for analyzing or intelligently applying that data.

This brings us to the third and most recent wave of database innovation.

## 2000s–2020s: NoSQL

The development of database technologies from the 2000s to the 2020s, approximately, is characterized as the advent of the NoSQL (non-SQL or "not only SQL") movement. The objective of this era was to create scalable technologies that stored, managed, and queried all shapes of data.

One way to describe the NoSQL era relates database innovation to the burgeoning of the craft beer market in the United States. The process of fermenting the beer didn't change, but flavors were added and the quality and freshness of ingredients were elevated. A closer connection developed between the brew master and the consumer,

yielding an immediate feedback loop on product direction. Now, instead of three brands of beer in your supermarket, you likely have more than 30.

Instead of finding new combinations for fermentation, the database industry experienced exponential growth in choices for data management technologies. Architects needed scalable technologies to address the different shapes, volumes, and requirements of their rapidly growing applications. Popular data shapes that emerged during this movement were key-value, wide-column, document, stream, and graph.

The message of the NoSQL era was quite clear: storing, managing, and querying data at scale in tables doesn't work for everything, just like not everyone wants to drink a light pilsner.

There were a few motivations that led to the NoSQL movement. These motivations are integral to understanding why and where we are within the hype cycle of the graph technology market. The three we like to call out are the need for data serialization standards, specialized tooling, and horizontal scalability.

First, the rise in popularity of web-based applications created natural channels for passing data between these applications. Through these channels, innovators developed new and different standards for data serialization such as XML, JSON, and YAML.

Naturally, these standardizations led to the second motivation: specialized tooling. The protocols for exchanging data across the web created structures that were inherently *not* tabular. This demand led to the innovation and rise in popularity of key-value, document, graph, and other specialized databases.

Last, this new class of applications came with an influx of data that put pressure on system scalability like never before. Derivatives and applications of Moore's law predicted the silver lining of this era as we saw the cost of hardware, and thus the cost of data storage, continue to decrease. The effects of Moore's law enabled data duplication, specialized systems, and overall computation power to become less expensive.[4]

Together, the innovations and new demands of the NoSQL era paved the way for the industry's migration from scale-up systems to scale-out systems. A scale-out system adds physical or virtual machines to increase the overall computational capacity of a system. A scale-out system, generally referred to as a "cluster," appears to the end-user as a single platform; the user has no idea that their workload is actually being served by a collection of servers. On the other hand, a scale-up system procures more powerful machines. Out of room? Get a bigger box, which is more expensive, until there are no bigger boxes to get.

4 Clair Brown and Greg Linden, *Chips and Change: How Crisis Reshapes the Semiconductor Industry* (Cambridge: MIT Press, 2011).

> Scaling out means adding more resources to spread out a load, typically in parallel. Scaling up means making a resource bigger or faster so that it can handle more load.

Given these three motivations, this versatile tool set for building scalable data architectures for nontabular data evolved to be the most important deliverable of the NoSQL era. Now development teams have choices to evaluate when designing their next application. They can select from a suite of technologies to accommodate different shapes, velocities, and scalability requirements of their data. There are tools that manage, store, search, and retrieve document, key-value, wide-column, and/or graph data at any scale. With these tools, we began working with multiple forms of data in ways previously unachievable.

What can we do with this unique collection of tools and data? We can solve more complex problems faster and at a larger scale.

## 2020s–?: Graph

We promised you that our history tour would be brief and purposeful. This section delivers on that promise by connecting the important moments from our condensed tour. Together, the connections we see across our industry's history set the stage for the fourth era of database innovation: the wave of graph thinking.

This era in innovation is shifting from efficiency of the storage systems to extracting value from the data the storage systems contain.

### Why the 2020s?

Before we can outline our perspective on the graph era, you might be wondering why we are starting the era of graph thinking in 2020. We want to take a brief moment to explain our position on the timing of the graph market.

Our callout to the general timeline of 2020 comes from the intersection of two trains of thought. At this intersection, we are crossing Geoffrey Moore's popular adoption model[5] with the timing observed during the past three eras of database innovation.

5 Geoffrey A. Moore and Regis McKenna, *Crossing the Chasm* (New York: HarperBusiness, 1999).

Like CODASYL, the technology adoption life cycle commonly attributed to Moore originated in the 1950s. See Everett Roger's 1962 book *Diffusion of Innovations*.[6]

Specifically, there is a proven and observable time lag between early adopters and the wide adoption of new technologies. We saw this time lag in "1980s–2000s: Entity-Relationship" on page 4 with relational databases during the 1970s. There was a 10-year lag between the first paper and corresponding viable implementations of relational technology. You can find examples of the same time lag within each of the other eras.

History has shown us that every era prior to the graph era contained a niche period that saw wide adoption years later. By looking to the 2020s, we are making this same assumption about the state of the graph market. History has also shown us that this doesn't mean that the existing tools are going to go away.

However you would like to measure it, this is not a stock market prediction where we are nailing down a date. Our outlook ultimately describes a new era of technology adoption that is being driven by an evolution of value. That is, value is shifting from efficiency to being derived from highly connected data assets. These changes take time and do not run on schedules.

#### Connecting the dots

Recall the three patterns of retrieval envisioned by the CODASYL committee in the 1960s: accessing data by keys, scans, and links. Extracting a piece of data by its key, in any shape, remains the most efficient way to access it. This efficiency was achieved during the entity-relationship era and remains a popular solution.

As for the second goal of the CODASYL committee, accessing data through scans, the NoSQL era created technologies capable of handling large scans of data. Now we have software and hardware capable of processing and extracting value from massive datasets at immense scale. That is to say: we have the committee's first two goals nailed down.

Last on the list: accessing data by traversing links. Our industry has come full circle.

The industry's return to focusing on graph technologies goes hand in hand with our shift from efficiently managing data to needing to extract value from it. This shift doesn't mean we no longer need to efficiently manage data; it means we have solved

6 Everett M. Rogers, *Diffusion of Innovations* (New York: Simon and Schuster, 2010).

one problem well and are moving on to address the harder problem. Our industry now emphasizes value alongside speed and cost.

Extracting value from data can be achieved when you are able to connect pieces of information and construct new insights. Extracting value in data comes from understanding the complex network of relationships within your data.

This is synonymous with recognizing the complex problems and complex systems that are observable across the inherent network in your data.

Our industry's and this book's focus looks toward developing and deploying technologies that deliver value from data. As in the relational era, a new way of thinking is required to understand, deploy, and apply these technologies.

A shift in mindset needs to occur in order to see the value we are talking about here. This mindset is a shift from thinking about your data in a table to prioritizing the relationships across it. This is what we call graph thinking.

## What Is Graph Thinking?

Without explicitly stating it, we already walked through what we call *graph thinking* during the whiteboard scene at the beginning of this chapter.

When we illustrated the realization that your data could look like a graph, we were recreating the power of graph thinking. It is that simple: graph thinking encompasses your experience and realizations when you see the value of understanding relationships across your data.

> Graph thinking is understanding a problem domain as an interconnected graph and using graph techniques to describe domain dynamics in an effort to solve domain problems.

Being able to see graphs across your data is the same as recognizing the complex network within your domain. Within a complex network, you will find the most complex problems to solve. And most high-value business problems and opportunities are complex problems.

This is why the next stage of innovation in data technologies is shifting from a focus on efficiency to a focus on finding value by specifically applying graph technologies.

## Complex Problems and Complex Systems

We have used the term *complex problem* a few times now without providing a specific description. When we talk about complex problems, we are referring to the networks within complex systems.

*Complex problems*
: Complex problems are the individual problems that are observable and measurable within complex systems.

*Complex systems*
: Complex systems are systems composed of many individual components that are interconnected in various ways such that the behavior of the overall system is not just a simple aggregate of the individual components' behavior (called "emergent behavior").

Complex systems describe the relationships, influences, dependencies, and interactions among the individual components of real-world constructs. Simply put, a complex system describes anything where multiple components interact with each other. Examples of complex systems are human knowledge, supply chains, transportation or communication systems, social organization, earth's global climate, and the entire universe.

Most high-value business problems are complex problems and require graph thinking. This book will teach you the four main patterns—neighborhoods, hierarchies, paths, and recommendations—used to solve complex problems with graph technology for businesses around the world.

## Complex Problems in Business

Data is no longer just a by-product of doing business. Data is increasingly becoming a strategic asset in our economy. Previously, data was something that needed to be managed with the greatest convenience and the least cost to enable business operation. Now it is treated as an investment that should yield a return. This requires us to rethink how we handle and work with data.

For example, the late stage of the NoSQL era saw the acquisitions of LinkedIn and GitHub by Microsoft. These acquisitions gave measurement to the value of data that solves complex problems. Specifically, Microsoft acquired LinkedIn for $26 billion on an estimated $1 billion in revenue. GitHub's acquisition set the price at $7.8 billion on an estimated $300 million in revenue.

Each of these companies, LinkedIn and GitHub, owns the graph to its respective networks. Their networks are the professional and the developer graphs, respectively. This puts a 26× multiplier on the data that models a domain's complex system. These

two acquisitions begin to illustrate the strategic value of data that models a domain's graph. Owning a domain's graph yields significant return on a company's valuation.

We do not want to misrepresent our intentions with these statistics. Observing high revenue multiples for fast-growing startups isn't a novelty. We specifically mention these two examples because GitHub and LinkedIn found and monetized value from data. These revenue multiples are higher than those valuations for similarly sized and similarly growing startups because of the data asset.

By applying graph thinking, these companies are able to represent, access, and understand the most complex problem within their domain. In short, these companies built solutions for some of the largest and most difficult complex systems.

Companies that have a head start on rethinking data strategies are those that built technology to model their domains' most complex problems. Specifically, what do Google, Amazon, FedEx, Verizon, Netflix, and Facebook all have in common? Aside from being among today's most valued companies, each one owns the data that models its domain's largest and most complex problem. Each owns the data that constructs its domain's graph.

Just think about it. Google has the graph of all human knowledge. Amazon and FedEx contain the graphs of our global supply chain and transportation economies. Verizon's data builds up our world's largest telecommunications graph. Facebook has the graph of our global social network. Netflix has access to the entertainment graph, modeled in Figure 1-2 and implemented in the final chapters of this book.

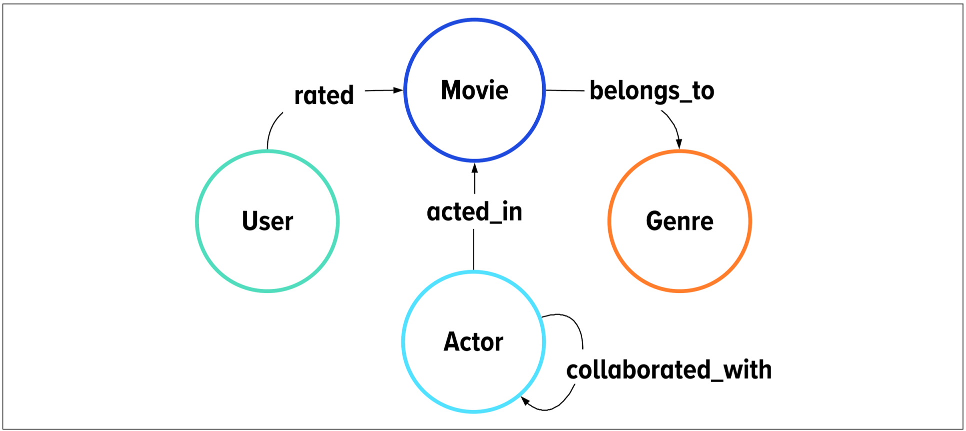



*Figure 1-2. One way to model Netflix's data as graph and the final example you will implement in this book: collaborative filtering at scale*

Going forward, those companies that invest in data architectures to model their domains' complex systems will join the ranks of these behemoths. The investment in technologies for modeling complex systems is the same as prioritizing the extraction of value from data.

If you want to get value out of your data, the first place to look is within its interconnectivity. What you are looking for is the complex system that your data describes. From there, your next decisions center around the right technologies for storing, managing, and extracting this interconnectivity.

## Making Technology Decisions to Solve Complex Problems

Whether or not you work at one of the companies previously mentioned, you can learn to apply graph thinking to the data in your domain.

So where do you get started?

The difficulty with learning and applying graph thinking begins with recognizing where relationships do or do not add value within your data. We use the two images in this section to simplify the stops along the way and illustrate the challenges ahead.

Though simple, Figure 1-3 challenges you to evaluate pivotal questions about your data. This first decision requires your team to know the type of data your application requires. We specifically start with this question because it is often overlooked.

Other teams before yours have overlooked the choices shown in Figure 1-3 because the lure of the new distracted them from following established processes for building production applications. This strain between new and established caused early teams to move too quickly through a critical evaluation of their application's goals. Because of this, we saw many graph projects fail and be shelved.

Let's step through what we mean in Figure 1-3 to keep you from repeating the common mistakes of early adopters of graph technologies.

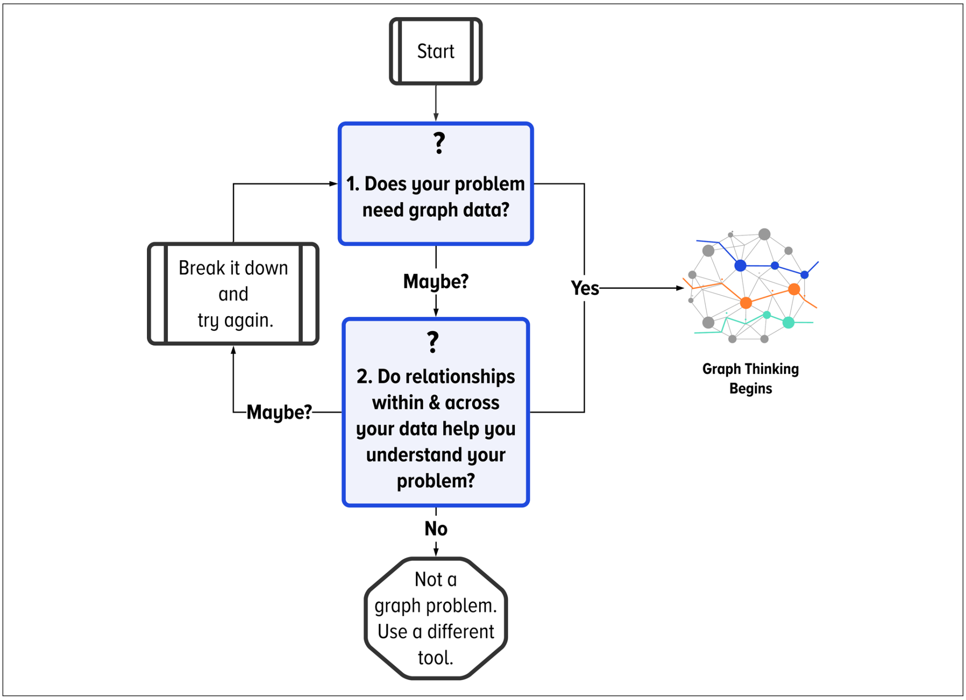



*Figure 1-3. Not every problem is a graph problem—the first decision you need to make*

### Question 1: Does your problem need graph data?

There are many ways of thinking about data. This first question in the decision tree challenges you to understand the shape of data that your application requires. For example, the mutual connections section on LinkedIn is a great example of a "yes" answer to question 1 in Figure 1-3. LinkedIn uses relationships between contacts so you can navigate your professional network and understand your shared connections. Presenting a section of mutual connections to an end user is a very popular way that graph shaped data is also used by Twitter, Facebook, and other social networking applications.

When we say "shape of data," we are referring to the structure of the valuable information you want to get out of your data. Do you want to know the name and age of a person? We would describe that as a row of data that would fit into a table. Do you want to know the chapter, section, page, and example in this book that shows you how to add a vertex to a graph? We would describe that as nested data that would fit into a document or hierarchy. Do you want to know the series of friends of your friends that connect you to Elon Musk? Here you are asking for a series of relationships that best fit into a graph.

Thinking top-down, we advise that the shape of your data drive the decision about your database and technology options. The types of data commonly used in modern applications are shown in Table 1-1.

*Table 1-1. An abbreviated summary of common data types, their shapes, and recommended types of databases*

| Data Description | Data Shape | Usage | Database Recommendation |
|---|---|---|---|
| Spreadsheets or tables | Relational | Retrieved by a primary key | RDBMS databases |
| Collections of files or documents | Hierarchical or nested | Root identified by an ID | Document databases |
| Relationships or links | Graph | Queried by a pattern | Graph databases |

For the most interesting data problems today, you need to be able to apply all three ways of thinking about your data. You need to be fluent in applying each to your data problem and its subproblems. For each piece of your problem, you need to understand the shape of the data coming into, residing within, and leaving your application. Each of these points, and any time in which data is in flight, drives the requirements for technology choices in your application.

If you are unsure about the shape of data that your problem requires, the next question from Figure 1-3 challenges you to think about the importance of relationships within your data.

#### Question 2: Do relationships within your data help you understand your problem?

The more pivotal question from Figure 1-3 asks whether relationships within your data exist and bring value to your business problem. A successful use of graph technology hinges on applying this second question from the decision tree. To us, there are only three answers to this question: yes, no, or maybe.

If you can confidently answer yes or no, then the path is clear. For example, LinkedIn's mutual connection section exemplifies a clear "yes" for graph-shaped data whereas LinkedIn's search box requires faceted search functionality and is a clear "no." We can make these clear distinctions by understanding the shape of data required to solve the business problem.

If relationships within your data help solve your business problem, then you need to use and apply graph technologies within your application. If they do not, then you

need to find a different tool. Maybe a choice from Table 1-1 will be a solution for your problem at hand.

The tricky part comes into play when you aren't exactly sure whether relationships are important to your business problem. This is shown with the "Maybe?" choice at left in Figure 1-3. In our experience, if your line of thinking brings you to this decision point, then you are likely trying to solve too large of a problem. We advise that you break down your problem and start back at the top of Figure 1-3. The most common problem we advise teams to break down is *entity resolution*, or knowing who-is-who in your data. Chapter 11 details an example of when to use graph structure within entity resolution.

#### Common missteps in understanding your data

Sometimes, seeing the shape of your data as a graph can subsume the importance of the other two data shapes: nested and tabular. Teams commonly misinterpret this red herring.

While you may think about your problem as a complex problem and therefore employ graph thinking to make sense of it, that does not mean you have to apply graph technologies to all data components of your problem. In fact, it may be advantageous to project certain components or subproblems onto tables or nested documents.

It will always be useful to think in projections (to files or tables). So our thought exercise in Figure 1-3 is more than "Which is the best way to think about your data?" It is above delving into a more agile thought process to break down complex problems into smaller components. That is, we encourage you to consider the best way to think about your data *for the current problem at hand*.

The shortest version of what we are trying to say in Figure 1-3 is: use the right tool for the problem at hand. And when we say "tool" here, we are thinking very broadly. We aren't necessarily using that term to refer to the choice of databases; we are thinking more broadly about the scope of data representation choices.

## So You Have Graph Data. What's Next?

The first question from Figure 1-3 challenges you to apply query-driven design to your data representation decisions. There may be parts of your complex problem that are best represented with tables or nested documents. That is expected.

But what happens when you have graph data and need to use it? This brings us to the second part of our graph thinking thought process, shown in Figure 1-4.

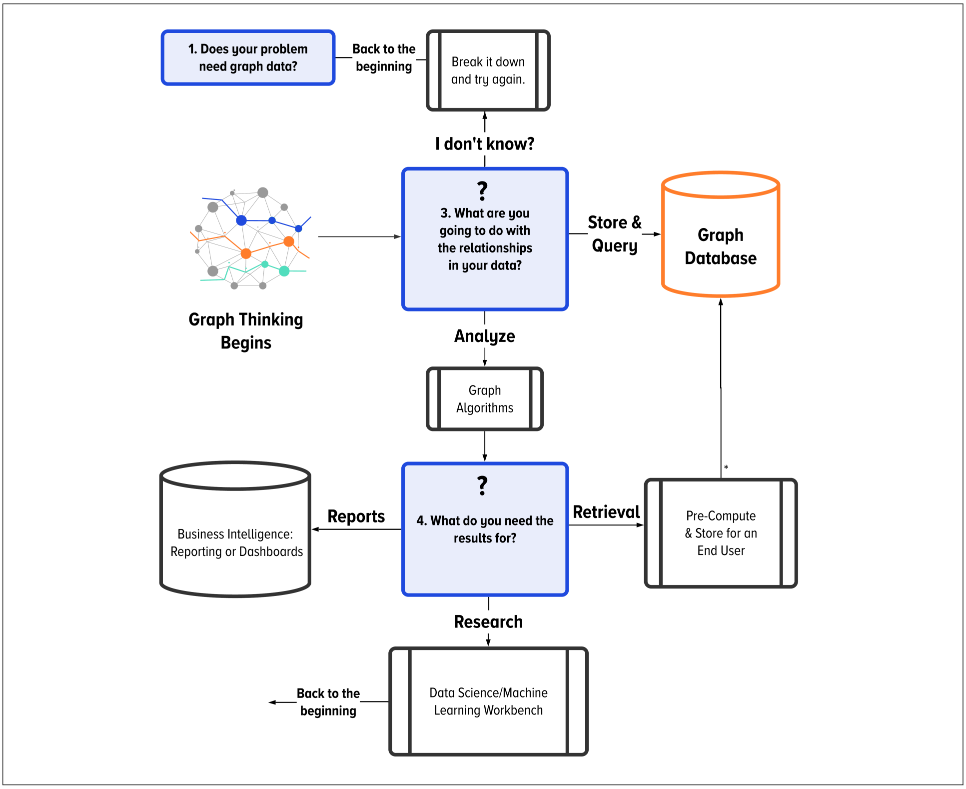



*Figure 1-4. How to navigate the applicability and usage of graph data within your application*

Moving forward, we are assuming that your application benefits from understanding, modeling, and using the relationships within your data.

## Question 3: What are you going to do with the relationships in your data?

Within the world of graph technologies, there are two main things you will need to do with your graph data: analyze it or query it. Continuing the LinkedIn example, the mutual connections section is an example of when graph data is queried and loaded into view. LinkedIn's research team probably tracks the average number of connections between any two people, which is an example of *analyzing* graph data.

The answer to this third question divides graph technology decisions into two camps: data analysis versus data management. The center of Figure 1-4 shows this question and the decision flow for each option.

> When we say *analyze*, we are referring to when you need to examine your data. Usually, teams spend time studying the relationships within their data with the goal of finding which relationships are important. This process is different from *querying* your graph data. *Query* refers to when you need to retrieve data from a system. In this case, you know the question you need to ask and the relationships required to answer the question.

Let's start with the option that moves to the right: the cases when you know your end application needs to store and query the relationships within your data. Admittedly, this is the least likely path today, due to the stage and age of the graph industry. But in these cases, you are primed and ready to move directly to using a graph database within your application.

From our collaborations, we have found a common set of use cases in which databases are needed to manage graph data. Those use cases are the topics of the upcoming chapters, and we will save them for later discussion.

Most often, however, teams know that their problems require graph-shaped data, but they do not know exactly how to answer their questions or which relationships are important. This is pointing toward needing to analyze your graph data.

From here, we challenge you and your team to take one more step in this journey. Our request is that you think about the deliverables from analyzing your graph data. Creating structure and purpose around graph analysis helps your team make more informed choices for your infrastructure and tools. This is the final question posed in Figure 1-4.

### Question 4: What do you need the results for?

Topics in graph data analysis can range from understanding specific distributions across the relationships to running algorithms across the entire structure. This is the area for algorithms such as connected components, clique detection, triangle counting, calculating a graph's degree distribution, page rank, reasoners, collaborative filtering, and many, many others. We will define many of these terms in upcoming chapters.

We most often see three different end goals for the results of a graph algorithm: reports, research, or retrieval. Let's dig into what we mean by each of those options.

> We are going into detail for all three options (reports, research, and retrieval) because this is what most people are doing with graph data today. The remaining technical examples and discussion in this book are focused primarily on when you have decided you need a graph database.

First, let's talk about reporting. Our use of the word *reports* refers to the traditional need for intelligence and insights into your business's data. This is most commonly referred to as business intelligence (BI). While debatably misapplied, the deliverables of many early graph projects aimed to provide metrics or inputs into an executive's established BI pipeline. The tools and infrastructure you will need for augmenting or creating processes for business intelligence from graph data deserve their own book and deep dive. This book does not focus on the architecture or approaches for BI problems.

Within the realm of data science and machine learning, you find another common use of graph algorithms: general research and development. Businesses invest in research and development to find the value within their graph-shaped data. There are a few books that explore the tools and infrastructure you will need for researching graph-structured data; this book is not one of them.

This brings us to the last path, labeled "retrieval." In Figure 1-4, we are specifically referencing those applications that provide a service to an end user. We are talking about data-driven products that serve your customers. These products come with expectations around latency, availability, personalization, and so on. These applications have different architectural requirements than applications that aim to create metrics for an internal audience. This book will cover these topics and use cases in the coming technology chapters.

Think back to our mention of LinkedIn. If you use LinkedIn, you have likely interacted with one of the best examples we can think of to describe the "retrieval" path in Figure 1-4. There is a feature in LinkedIn that describes how you are connected to any other person in the network. When you look at someone else's professional profile, this feature describes whether that person is a 1st-degree, 2nd-degree, or 3rd-degree connection. The length of the connection between you and anyone else on LinkedIn tells you useful information about your professional network. This LinkedIn feature is an example of a data product that followed the retrieval path of Figure 1-4 to deliver a contextual graph metric to the end users.

The lines between these three paths can be blurry. The difference lies between building a data-driven product or needing to derive data insights. Data-driven products deliver unique value to your customers. The next wave of innovation for these products will be to use graph data to deliver more relevant and meaningful experiences. These are the interesting problems and architectures we want to explore throughout this book.

#### Break it down and try again

Occasionally you may respond to the questions throughout Figure 1-3 and Figure 1-4 with "I don't know"—and that is OK.

Ultimately, you are likely reading this book because your business has data and a complex problem. Such problems are vast and interdependent. At your problem's highest level, navigating the thought process we are presenting throughout Figure 1-3 and Figure 1-4 can seem out of touch with your complex data.

However, drawing on our collective experience helping hundreds of teams around the world, our advice remains that you should break down your problem and cycle through the process again.

Balancing the demands of executive stakeholders, developer skills, and industry demands is extremely difficult. You need to start small. Build a foundation upon known and proven value to get you one step closer to solving your complex problem.

> What happens if you ignore making a decision? Too often, we have seen great ideas fail to make the transition from research and development to a production application: the age-old analysis paralysis. The objective of running graph algorithms is to determine how relationships bring value to your data-driven application. You will need to make some difficult decisions about the amount of time and resources you spend in this area.

## Seeing the Bigger Picture

The path to understanding the strategic importance of your business's data is synonymous with finding where (and whether) graph technology fits into your application. To help you determine the strategic importance of graph data for your business, we have walked through four very important questions about your application development:

1. Does your problem need graph data?
2. Do relationships within your data help you understand your problem?
3. What are you going to do with the relationships in your data?
4. What do you need to do with the results of a graph algorithm?

Bringing these thought processes together, Figure 1-5 combines all four questions into one chart.

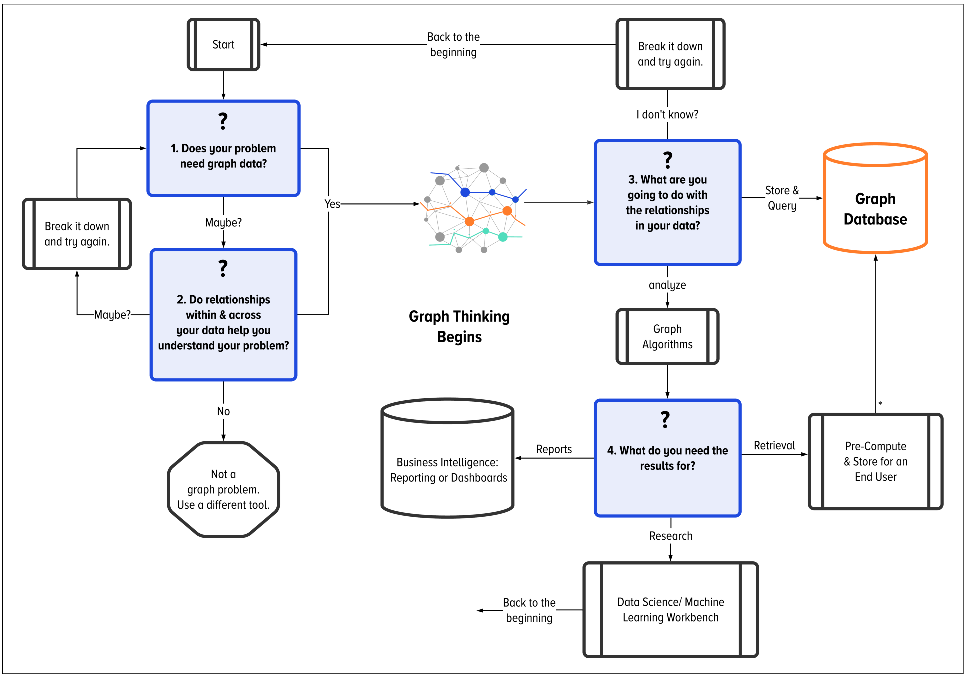



*Figure 1-5. The decision process that sparked the creation of this book: how to navigate the applicability and usage of graph technology within your application*

We spent time walking through the entire decision tree for two reasons. First, the decision tree depicts a complete picture of the thought process we use when we build, advise on, and apply graph technologies. Second, the decision tree illustrates where this book's purpose fits into the space of graph thinking.

That is, this book serves as your guide to navigating graph thinking in the paths throughout Figure 1-5 that end in needing a graph database.

# Getting Started on Your Journey with Graph Thinking

When properly leveraged, your business's data can be a strategic asset and an investment that yields a return. Graphs are of particular importance here since network effects are a powerful force that provides exquisite competitive advantage. Additionally, today's design thinking encourages architects to view their business's data as something that needs to get managed with maximal convenience and minimal cost.

This mindset requires a rethinking of how we handle and work with data.

Changing a mindset is a long journey, and any journey begins with one step. Let's take that step together and learn the new set of terms we will be using along the way.

CHAPTER 2

# Evolving from Relational to Graph Thinking

Together over the years, we have advised hundreds of teams on where and how to get started with graph data and graph technologies. From our conversations with those teams, we assembled the most common questions and advice for introducing graph thinking and graph data into your business.

We want to start your journey toward graph thinking with the following three questions that every team will encounter when evaluating graph technologies:

1. Is graph technology better for my problem than relational technology?
2. How do I think about my data as a graph?
3. How do I model a graph's schema?

Those teams that spend the time up-front to understand these three topics are more likely to successfully integrate graph technologies into their stack. Conversely, from our experience, businesses shelved early-stage graph projects because their teams skipped through collectively understanding these questions for their business.

## Chapter Preview: Translating Relational Concepts to Graph Terminology

The three questions in the opening section form the outline of this chapter.

We will start off with an abbreviated tour of the differences between relational and graph technologies. Then we will walk through an abbreviated tour of relational data modeling. From the model, we will translate the relational concepts to graph modeling techniques and take a short tour of some fundamental terms from graph theory.

We will also introduce the Graph Schema Language (GSL), a language (or tool) that helps you translate a visual graph schema into code. We created the GSL to help you answer questions 2 and 3 from the beginning of the chapter. Throughout this book, we will use the GSL as a teaching tool to translate a diagram into schema statements.

Inevitably, you are going to have to make some tough decisions about whether, where, and how to introduce graph thinking and technology into your workflow. In this chapter, we are going to introduce tools and techniques to help you navigate a large pool of technical opinions. The foundations we provide here will help you evaluate whether graph technology is the right choice for your next application.

The concepts and technology decisions introduced in this chapter will serve as the foundational material for our future examples. We are using this chapter to clearly illustrate the vocabulary that we will use to describe graph database schema and graph data in the examples throughout this book.

The introduction of graph data into your application brings a new paradigm of thinking about what is important within your data. Understanding the differences in these principles starts with evolving your mindset from relational to graph thinking.

# Relational Versus Graph: What's the Difference?

So far we have mentioned two different technologies: relational and graph. When we talk about relational systems, we are referring to organizing your data in a way that focuses on the storage and retrieval of real-world entities such as people, places, and things. When we talk about graph systems, we are referring to systems that focus on the storage and retrieval of relationships. These relationships represent the connections between real-world entities: people know people, people live in places, people own things, and so on.

Both systems can represent entities and relationships alike, but are built and optimized for one over the other.

The line between selecting a relational system or selecting a graph system for your application is gray; each choice has benefits and drawbacks. Choosing between a relational database and a graph database typically generates a conversation about storage requirements, scalability, query speed, ease of use, and maintainability. While any aspect of such a conversation is worth discussing, we aim to shed light on the more subjective criteria: ease of use and maintainability.

> Even though the words relational and relationships are very similar, we use them explicitly to refer to two different types of technologies. The word *relational* describes a type of database, like Oracle, MySQL, PostgreSQL, or IBM Db2. These systems were created to apply a specific field of mathematics to data organization and reasoning—namely, relational algebra. On the other hand, we use the word *relationship* solely in reference to graph data and graph technologies. These systems were created to apply a different field of mathematics to data organization and reasoning—namely, graph theory.

Choosing between relational and graph technologies can be difficult because you cannot compare them at a feature-functionality level. Their differences can be traced to their cores as a result of their being built on distinct mathematical theories: relational algebra for relational systems and graph theory for graph systems. That means the suitability of each technology depends to a large degree on the applicability of those theories and their associated lines of thinking to your problem.

We are going to drill a little further into the differences between relational and graph technologies in the following sections, for two reasons. First, since most people are familiar with relational thinking, we can introduce graph thinking in contrast to relational thinking. Second, we want to provide a response to the inevitable question, "Why not just use an RDBMS?" Both of these reasons are important to explore in the context of understanding graph technology because relational systems are very mature and widely adopted.

Throughout this book, we will use data to illustrate concepts, examples, and new terminology. Let's start with the data that we will be using in this chapter to illustrate the differences between relational and graph concepts. You will see this data in the example that spans Chapters 3, 4, and 5.

## Data for Our Running Example

We will use the data in Table 2-1 to construct relational and graph data models.

For our first use case, the data describes several customers' assets in the financial services industry. The customers can share accounts and loans, but a credit card can be used by only one customer.

Let's look at a few rows of the data. Table 2-1 displays data about five customers. These five customers and their data will be used to build data models and illustrate new concepts throughout this chapter and the next three chapters.

*Table 2-1. A sample of the data created to illustrate the concepts, examples, and terminology in the next two chapters*

| customer_id | name | acct_id | loan_id | cc_num |
|---|---|---|---|---|
| customer_0 | Michael | acct_14 | loan_32 | cc_17 |
| customer_1 | Maria | acct_14 | none | none |
| customer_2 | Rashika | acct_5 | none | cc_32 |
| customer_3 | Jamie | acct_0 | loan_18 | none |
| customer_4 | Aaliyah | acct_0 | [loan_18, loan_80] | none |

There are five unique customers in the five rows of sample data shown in Table 2-1. Some of these customers share accounts or loans to illustrate different types of users we typically see in a financial services system.

For example, `customer_0` and `customer1`, or Michael and Maria, represent a typical parent-child relationship; Michael is the parent, and Maria is the child. The data about `customer_2`, Rashika, indicates they are a sole user of this financial service. We usually see the highest volume of this type of user in large applications; customers like Rashika only have data that is unique to the customer and is not shared by anyone else. Last, `customer_3` and `customer_4` (Jamie and Aaliyah) share an account and a loan. This type of data typically indicates that the users are partners who have joined their financial accounts.

If this were your company's sample data, imagine the conversation you might have with your coworker about modeling this data. In this scenario, you are sharing a whiteboard, or other illustrative tool, and you are trying to map out the entities, attributes, and relationships within the data. Whether or not you use a relational or graph system, you likely would be having a discussion similar to the conceptual model in Figure 2-1.

From Table 2-1, we find four main entities: customers, accounts, loans, and credit cards. These entities each have relationships tied to the customer. Customers can have multiple accounts, and those accounts can have more than one customer. Customers can also have multiple loans, and those loans can have more than one customer. Finally, customers can have multiple credit cards, but each credit card is unique to one customer.

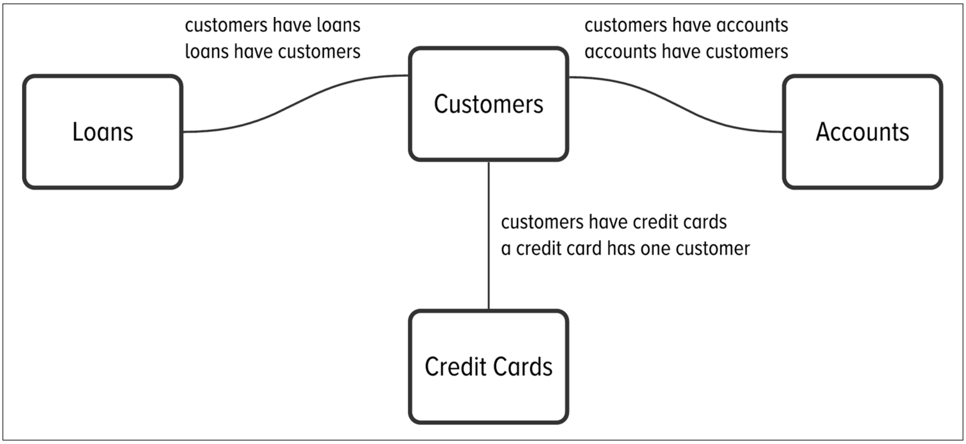



*Figure 2-1. A conceptual description of the relationships observed in the data in Table 2-1*

# Relational Data Modeling

Your transition from relational to graph thinking starts with data modeling. Understanding data modeling in these two systems begins to illustrate why graph technologies can be a better fit.

For anyone who has been a database practitioner, you've probably been introduced to visual ways of modeling data in a relational system. The most popular choices for creating relational data models are to use the *unified modeling language* (UML) or to use *entity-relationship diagrams* (ERDs).

In this section, we will use the example data from Table 2-1 to complete an abbreviated walk-through of relational data modeling with an ERD. We have included just enough information in this section to provide a first step from relational to graph thinking. This is not intended to be used as a full introduction into the world of relational data modeling. We recommend the seasoned book by C. Batini et al.[1] for complete details on relational data modeling. And for those of you who are very comfortable with third normal form, you can skip this next bit and head directly to "The Graph Schema Language" on page 33.

1 Carlo Batini, Stefano Ceri, and Shamkant B. Navathe, *Conceptual Database Design: An Entity-Relationship Approach*, vol. 116 (Redwood City, CA: Benjamin/Cummings, 1992).

## Entities and Attributes

Generally speaking, data modeling techniques help you describe the real world by describing the entities and their attributes within your data. Each of those concepts has a specific meaning:

*Entity*
: An entity is an object such as a person, place, or thing that you need to track in your database.

*Attribute*
: An attribute refers to a property of an entity such as names, dates, or other descriptive features.

The traditional approach for relational data modeling starts with identifying the entities (people, places, and things) in your data and the attributes (names, identifiers, and descriptions) of those entities. Entities could be customers, bank accounts, or products. Attributes are concepts such as a person's name or bank account number.

For this exercise in data modeling, let's start by modeling two entities from Table 2-1: customers and bank accounts. In a relational system, we traditionally view the entities as tables. This is illustrated in Figure 2-2.

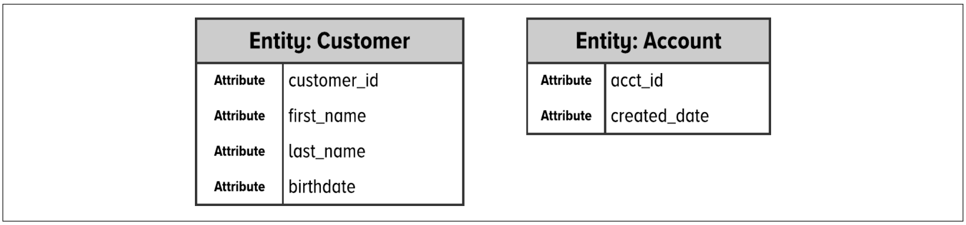



*Figure 2-2. The traditional approach to modeling the data within your application: identifying the entities and their attributes*

There are two main concepts shown in Figure 2-2: entities and their respective attributes. There are two entities in this diagram: customers and accounts. For each entity, there is a list of attributes that describe the entity. A customer can be described by a unique identifier, name, birthdate, and so on. There are also descriptive attributes for accounts: a unique account identifier and the date the account was created.

In a relational database, each entity becomes a table. The rows of the table contain sample data about that entity, and each column contains values for the descriptive attributes.

## Building Up to an ERD

In the real world, customers own accounts. The next step in designing a relational database would be to conceptually model this connection. We need to add to our model a way to describe how a person owns a bank account. A popular method for modeling the link from customers to accounts is shown in Figure 2-3.

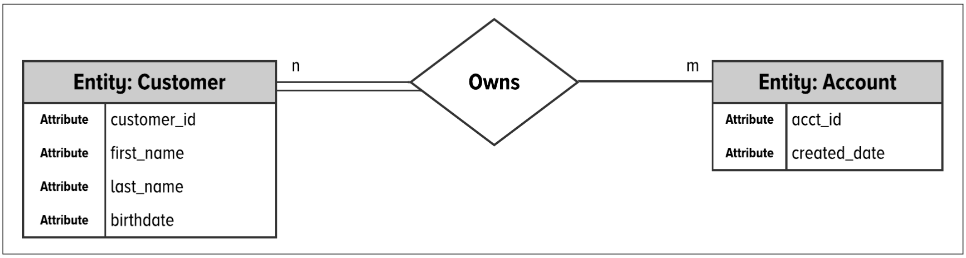
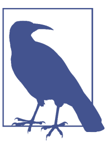



*Figure 2-3. An entity-relationship diagram for customers and bank accounts*

One visual element that we added between Figure 2-2 and Figure 2-3 is the diamond that connects the person and account entity tables. This connection indicates that there is a link between customers and accounts in the database. Namely, customers own accounts.

The image's other visual details include the double lines between the person table and the `owns` connection. Here we see an `n`, with an `m` on the opposite side of the `owns` connection. This notation indicates that this is a `many-to-many` connection between customers and accounts. Specifically, this translates to the idea that one person can own many accounts and that one account can be owned by many customers.

The following nuance about the implementation details is important: links that are shown in ERDs translate to tables or foreign keys. That is, the connections between customers and their accounts are stored as a table within a relational system. This means that the `owns` table essentially translates to another entity in the database.



Using tables to represent the connections within your data as entities makes it more difficult to understand the links within your data. The mental leap from natural understanding to tabular retrieval is a significant mental hurdle to overcome. This is especially true when you need to understand the connectedness of your data.

Even though we have been forced to think this way for decades, there are better ways.

Let's revisit the data from Table 2-1. However, this time we are going to use the data to illustrate concepts in graph data, followed by how we will model the data in a graph database.

# Concepts in Graph Data

We will use this section to introduce useful terminology from the graph theory community. These terms are used to describe the connectivity of the graph data. Let's visualize the graph data about the first three people from our sample data.

The data visualized in Figure 2-4 will be used to illustrate the fundamental concepts in the rest of this section. This data contains information about three people: Michael, Maria, and Rashika. Michael and Maria share an account, as seen in Figure 2-4. Rashika does not share any data with the other two customers in our example.

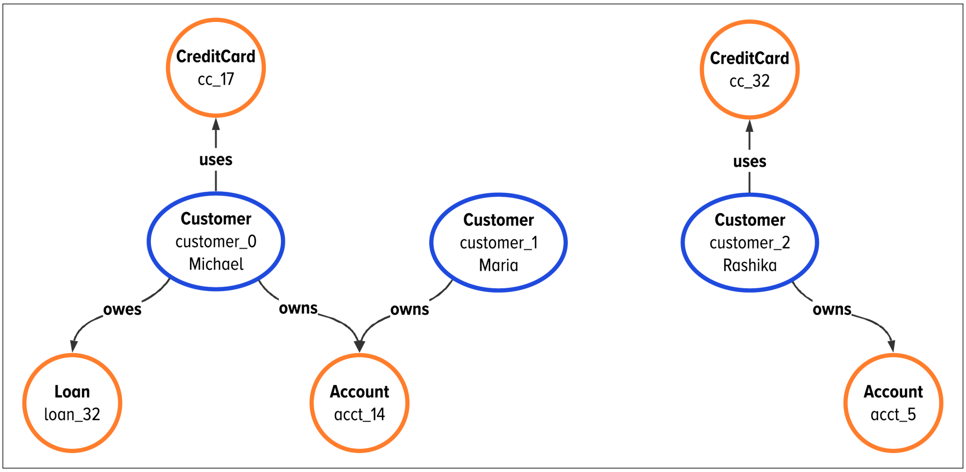



*Figure 2-4. A look at the graph data we will use to introduce new graph terminology in this chapter*

## Fundamental Elements of a Graph

The first concepts we need to introduce are the fundamental elements of graphs and graph data and their definitions. These terms are used across all members of the graph community and are accepted as the fundamental elements of a graph.

*Graph*
: A graph is a representation of data with two distinct elements: vertices and edges.

*Vertex*
: A vertex (pl. vertices) represents a concept or entity in data.

*Edge*
: An edge represents a relationship or link from one vertex to another.

You have already seen the fundamental elements we are talking about. Our financial data from Figure 2-4 contains four conceptual entities: customers, accounts, credit cards, and loans. These entities naturally translate into the vertices of our graph.

> We avoid the term *nodes* in this book because we are focusing on distributed graphs, and *nodes* has different meanings in distributed systems, graph theory, and computer science.

Next, we use edges to connect our vertices. These connections illustrate the relationships that exist between the pieces of data. In graph data, an edge connects two vertices as an abstract representation of a relationship between the two objects.

For this data, we will use edges to show the relationship between a person and their financial data. We model the data to say that the customer owns accounts, the customer owes loans, and the customer uses credit cards. The edges in the graph database become the relationships of `owns`, `owes`, and `uses`.

Together, all of the vertices and edges in the data represent the full `graph`.

## Adjacency

While there are many foundational topics in graph theory to explore, the term to start with is *adjacency*. You will find this term used throughout graph theory to talk about how data is connected. Essentially, adjacency is the mathematical term used to describe whether vertices are connected to each other. Formally, it is defined as follows:

*Adjacency*
: Two vertices are adjacent if they are connected by an edge.

In Figure 2-4, `Maria` is adjacent to `acct_14`. Also, we see that both `Michael` and `Maria` are adjacent to `acct_14` because they both own that account. The benefit to using graph data in your application is immediately apparent when you can see how different entities are related in a way that you may not have previously seen.

The idea of adjacency will come up many more times throughout this book, in topics ranging from the connectedness of data to different storage formats on disk. For now, it is important to know only that this popular term refers to how vertices are connected.

## Neighborhoods

Data that is connected forms communities. In graph theory, these communities are called *neighborhoods*.

*Neighborhood*
: For a vertex `v`, all vertices that are adjacent to `v` are said to be within the neighborhood of `v`, written `N(v)`. All vertices within a neighborhood are neighbors of `v`.

Figure 2-5 shows the concept of *graph neighborhoods* starting from `customer_0`, `Michael`. In this sample data, the vertices `cc_17`, `loan_32`, and `acct_14` are directly connected, or adjacent, to `Michael`. We call this the *first neighborhood* of `customer_0`.

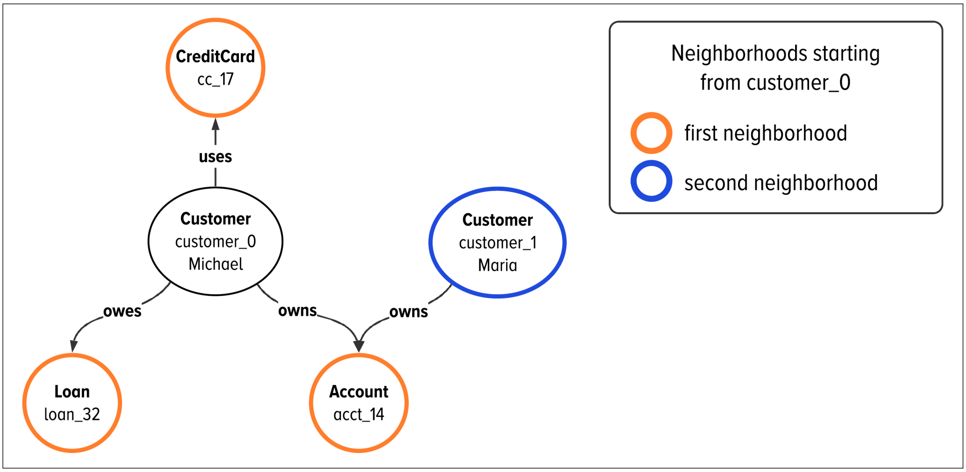



*Figure 2-5. A visual example of graph neighborhoods starting from `customer_0`*

You can continue this concept by walking further away from the starting vertex. The *second neighborhood* consists of those vertices that are two edges away from `Michael`; `Maria` is in the second neighborhood of `Michael`. It also works to say the reverse, that `Michael` is in the second neighborhood of `Maria`. This can continue on throughout the graph as we walk through the full depth of vertices from a singular starting point.

## Distance

The concept of neighborhoods brings us to distance. Another way to talk about the connectedness of this sample data is to say how many steps it takes to walk from one vertex to another. Talking about `Michael`'s first or second neighborhood is the same as finding all vertices that are a distance of 1 or 2 from `Michael`.

*Distance*
: In graph data, distance refers to the number of edges that you have to walk through to get from one vertex to another.

In Figure 2-5, we selected the starting point as the vertex `Michael`. The vertices `cc_17`, `loan_32`, and `acct_14` are in `Michael`'s first neighborhood, which is the same as a distance of 1 from `Michael`.

In mathematical communities, you will see this written as `dist(Michael, cc_17) = 1`. That also means that everything in the second neighborhood from your starting point is two edges away, and so on—specifically, `dist(Michael, Maria) = 2`.

## Degree

The ideas of adjacency, neighborhoods, and distance help us understand *if* two pieces of data are connected. For many applications, it is especially useful to understand *how well* a piece of data is connected to its neighbors.

The difference between if and how well data is connected introduces a new term from the math community: *degree*.

*Degree*
: A vertex's *degree* is the number of edges that are incident to (i.e., touch) the vertex.

In other words, we talk about a vertex's degree in reference to the number of edges that touch that vertex.

Recall Figure 2-4 as we walk through the upcoming examples. In that figure, we see that there are three edges that connect `Michael` to `cc_17`, `loan_32`, and `acct_14`. We apply this to say that the degree of `Michael` is three, or `deg(Michael) = 3`.

In this data, we have two vertices that have a degree of two. Specifically, `acct_14` is adjacent to `Michael` and `Maria`, so it has a degree of two. On the right side of the image, we see that `Rashika` also has only two edges. That means that `Rashika` has a degree of two.

There are a total of five vertices in our example data that have a degree of one. They are `loan_32`, `cc_17`, `Maria`, `cc_32`, and `acct_5`.

> In graph theory, a vertex with a degree of one is called a *leaf*.

We also break down a vertex's degree into two subcategories according to whether the edge starts at or ends with that particular vertex. Let's introduce two new terms that describe these categories.

*In-degree*
: A vertex's in-degree is the total number of *incoming* edges that are incident to (or touch) the vertex.

*Out-degree*
: A vertex's out-degree is the total number of *outgoing* edges that are incident to (or touch) the vertex.

Let's apply these definitions to the examples we just walked through.

All three of `Michael`'s edges start at `Michael` and end at other vertices: `cc_17`, `loan_32`, and `acct_14`. Therefore, we say that the out-degree of `Michael` is three because all three of the edges are outgoing.

The in-degree of `acct_14` is two because it has incoming edges from `Michael` and `Maria`. Both of `Rashika`'s edges are outgoing edges, so we say that `Rashika` has an out-degree of two.

The in-degree of `cc_17` is one because the edge is incoming; the same is true for `loan_32`, `cc_32`, and `acct_5`. `Maria` has an out-degree of one because its one edge is an outgoing edge to `acct_14`.

#### Implications of vertex degree

Data scientists and graph theorists use a vertex's degree to understand the type of connections found within the graph data. One of the places to start is to find the most highly connected vertices within your graph.

Depending on the application, vertices of very high degree can be thought of as hubs or highly influential entities.

It is useful to find these highly connected vertices because there are performance ramifications when they are stored or queried in a graph database. To a graph database practitioner, vertices of extremely high degree (>100,000 edges) are known as *supernodes*.

For the purposes of this section, we want to illustrate how to apply and interpret graph structure in your application. We will get into the performance details of highly connected vertices in Chapter 9, where we formally define supernodes, reason about their influence in your database, and step through mitigation strategies.

Recall that we opened this chapter with three questions. Everything up until now addressed the first two of those questions. The final section of this chapter introduces

a tool to teach you how to model graph schema by translating visual diagrams into code. Let's dive in.

# The Graph Schema Language

Graph practitioners, academics, and engineers have generally agreed on terms and methods for illustrating graph data. However, the terms are used confusingly across the technical and academic communities. There are words that mean one thing to a graph database practitioner and something different to a graph data scientist.

To address confusion across the communities, in this book we are introducing and formalizing terminology to describe graph schema. This language is called the Graph Schema Language, or GSL. The GSL is a visual language for applying concepts to create graph database schema.

We created the GSL as a teaching tool to use throughout the examples in this book. Our purpose in creating, introducing, and using the GSL is to normalize how graph practitioners communicate conceptual graph models, graph schema, and graph database design. To us, this set of terminology and visual illustrations complements the graph languages popularized by the academic community and the standardization initiatives within the graph community.

We will use the visual cues and terminology described in this section throughout the conceptual graph models used in this book. We hope the many upcoming examples are good practice for translating a visual schematic into schema code.

## Vertex Labels and Edge Labels

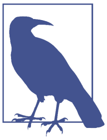
The fundamental elements of graph data—vertices and edges—give us the first terms of the Graph Schema Language (GSL): *vertex labels* and *edge labels*. Where relational models use *tables* to describe the data in Figure 2-3, we use *vertex labels* and *edge labels* to describe a graph's schema.

*Vertex label*
: A vertex label is a set of objects that are semantically homogeneous. That is, a vertex label represents a class of objects that share the same relationships and attributes.

*Edge label*
: An edge label names the type of relationship between vertex labels in your database schema.

In graph modeling, we label each entity with a vertex label and describe the relationship between entities with an edge label.

Generally speaking, vertex labels describe entities in your data that share attributes of the same type and relationships of the same label. Edge labels describe relationships between vertex labels.



The terms *vertices* and *edges* are used in reference to data. To describe a database's schema, we use the terms *vertex labels* and *edge labels*.

For the data from Table 2-1, we would model the same customer and account with the conceptual graph model shown in Figure 2-6. This conceptual graph model looks very similar to the ERD from Figure 2-3, but with the translation to using the first two terms of GSL.

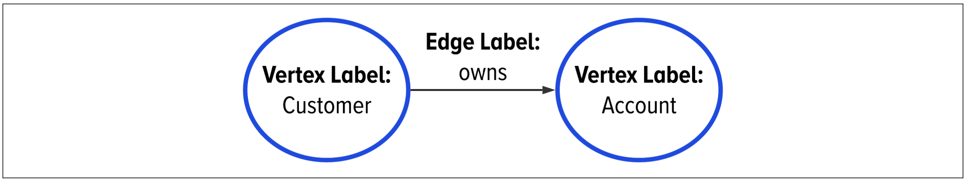



*Figure 2-6. A graph model showing the vertex and edge labels for customers and bank accounts*

In the GSL, a vertex label is illustrated with a circle containing the label's name. Figure 2-6 shows this for the `Customer` and `Account` vertex labels. An edge label is illustrated with a named line between two vertex labels. We see this in Figure 2-6 with the `owns` edge label between the `Customer` and `Account` vertex labels. When we look at this illustration, we infer that a customer has a relationship to an account; specifically, the customer owns the account. We will get into an edge's direction later, in "Edge Direction" on page 35.

## Properties

Where relational models use attributes to describe the data in Figure 2-3, properties describe data in graph modeling. That is, where we had attributes before, we have properties in a graph model.

*Property*
: A property describes features of a vertex label or an edge label, such as names, dates, or other descriptive features.

In Figure 2-7, each vertex label has a list of properties associated with it. These properties are the same attributes from the relational ERD in Figure 2-3. A customer vertex is described by its unique identifier, name, and birthdate. An account is described with its account ID and created date, as before. We add an edge label of `owns` to describe the relationship between the two entities in this data model.

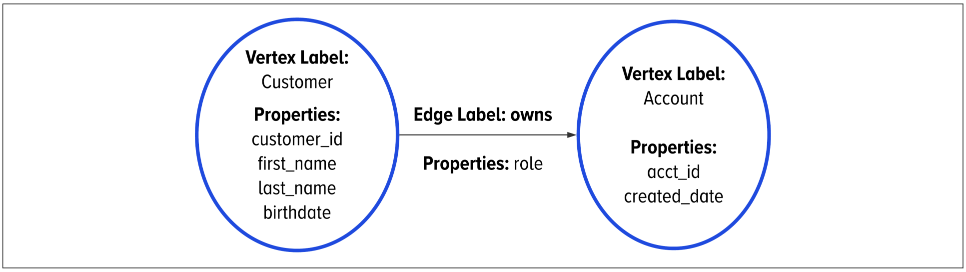



*Figure 2-7. A graph model for customers and bank accounts*

> Note that the term *property* applies to concepts in graph schema and graph data.

## Edge Direction

The next modeling concept defined in the GSL is an edge's direction. When we set up our edge labels in our data model, we connected the vertex labels together in a way that follows how we would naturally talk about the data; we would say that a customer owns an account, and we modeled the data in our graph that way. This also gives each edge a *direction*.

There are two ways in which you can model the direction of an edge: directed and bidirectional.

*Directed*
: A directed edge goes one way: from one vertex label to the other vertex label.

*Bidirectional*
: A bidirectional or bidirected edge goes in both directions between the vertex labels.

Using GSL, we indicate the direction of an edge with an arrow at either one end or both ends of the edge line. This is illustrated in Figure 2-8 with the arrow below the edge label.

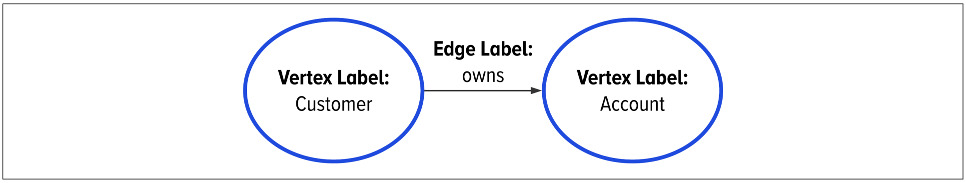



*Figure 2-8. The use of a line with a single arrow to indicate that an edge label is directed*

We would say that our example in Figure 2-8 shows a directed edge label. We have an edge label that goes from the customer to an account. This edge label uses a directed edge to model that a customer owns an account.

On the other hand, it might be useful to model our data in the opposite direction. One way to do this is by adding a second directed edge from the account to the customer. This edge indicates that the account is owned by the customer, like in Figure 2-9. We say that all of the edges in Figure 2-9 are *directed* because they go only one way.

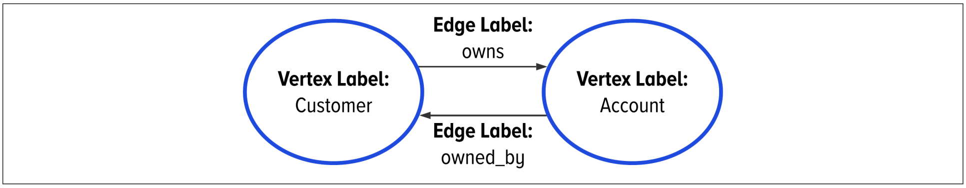



*Figure 2-9. The use of two different directed edge labels to allow walking in both directions between customers and accounts*

An edge label's direction comes from how we communicate about our data. When you describe your data, you use a subject, a predicate (i.e., a verb), and an object to communicate about your domain. To see this, consider how you would describe the sample data we have been using so far in this chapter. You likely are thinking something like "customers own accounts" or "bank accounts are owned by customers." In the first phrase, the subject is "customers," the predicate is "owns," and the object is "accounts." This gives us a source vertex label, `Customer`, and a destination vertex label, `Account`; the predicate "owns" translates to an edge label and has a direction from the customer to their account. We can follow a similar process for the second phrase to derive an edge label of "owned_by" from `Account` to `Customer`.

Loosely speaking, the identification of the subject, predicate, and object of your description creates an edge label's direction. The subject is the first vertex label and is where an edge starts. In the GSL, we call this the *domain*. Then the predicate becomes your edge label. Last, the object is the destination or *range* of an edge label. This means the edge label comes from the domain and goes to the range. This gives us two new terms:

*Domain*
: The domain of an edge label is the vertex label from which the edge label originates or starts.

*Range*
: The range of an edge label is the vertex label to which the edge label points or ends.

The last concept to illustrate in this section is a bidirectional edge. For the data we have talked about so far, it doesn't exactly make semantic sense to have an edge label that goes in both directions. That is, it is meaningful to say that a customer owns an account, but it doesn't make sense to say "an account owns a customer." We have to change the edge label to say "an account is owned by a customer."

To best illustrate an edge label that is bidirectional, let's add relationships between customers into our example. Specifically, let's add an edge label that connects customers who are family members. This is a better example to illustrate a bidirectional edge label and is shown in Figure 2-10.

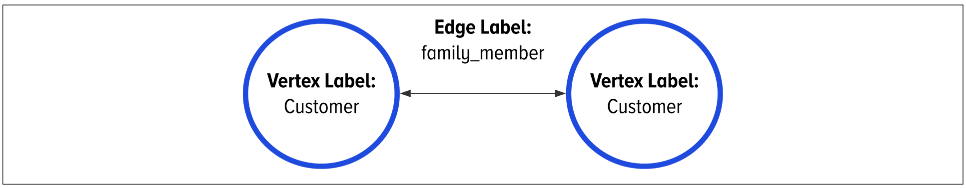



*Figure 2-10. The use of a line with two arrows to indicate that an edge label is bidirectional*

In this model, we are indicating that customers can be family members of other customers. We interpret this type of relationship as a reciprocal relationship: if you are a family member to someone else, they are also your family member. We model this in the GSL using a line with an arrow on both ends and say that this edge label is bidirectional or bidirected.

> In graph theory, a *bidirected* edge is the same thing as an undirected edge. That is, modeling a relationship in both directions is essentially the same as not having any specific direction. However, in the context of this book, we are using relationships between data to provide meaning to an application and therefore have to consider an edge's direction.

When first encountering direction, it can be a tricky concept to wrap your head around. In graph development, one of the best ways to think about direction comes from how you speak about your data. We recommend creating a description of your data and identifying how you would explain the relationships within it. This helps

mentally translate your conceptual understanding of your data's relationships into an edge label's direction.

## Self-Referencing Edge Labels

Without explicitly calling it out, we introduced a new concept in Figure 2-10 that we would like to define now. If an edge starts and ends on the same vertex label, we say that is a *self-referencing* edge label. In the GSL, we would draw and notate this as seen in Figure 2-11.

*Self-referencing*
: A self-referencing edge label is where the edge label's domain and range are the same vertex label.

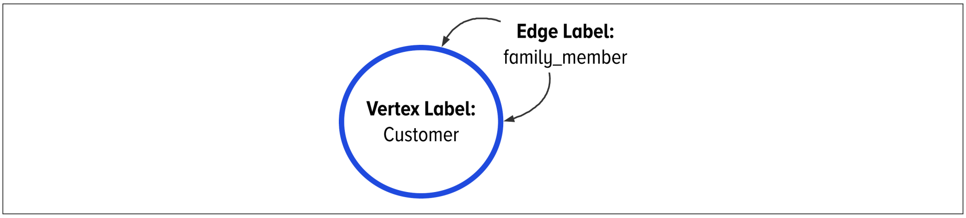



*Figure 2-11. The use of an edge label in the GSL to indicate a bidirectional, self-referencing edge*

Figure 2-11 is the correct way to illustrate an edge label that starts and ends on the same vertex label. We say that this is a self-referencing edge label. In the case of Figure 2-11, it is also a bidirected edge label. However, not all self-referencing edge labels are bidirected.

> You will see an example of a directed, self-referencing edge label in an upcoming chapter. This is the case when you need to model a recursive relationship—specifically, when something is contained within something else, or when you have a parent-child relationship.

## Multiplicity of Your Graph

When you start diving into data modeling with graphs, you will probably want a way to indicate how many relationships can exist between different vertex labels in your graph.

We have some good news on this topic. There is only one option for describing the number of relationships in most graph models: many.

In DataStax Graph and in most other graph databases, all edge labels represent many-to-many relationships. Meaning, any vertex can have many other vertices connected to it by a particular edge label. In an ERD, this is called `many-to-many`; in UML you use 0..* to 0..*. Sometimes, this is also referred to as an `m:n` relationship within the relational community.

We use the term *multiplicity* to describe this concept:

*Multiplicity*
: Multiplicity is a specification of the range of allowable sizes that a group may assume. Namely, multiplicity describes the range of allowable sizes that the group of vertices adjacent to a given vertex along a particular edge label may assume.[2]

> The actual size of the set or collection is referred to as *cardinality*. Cardinality is defined as the finite number of elements in a particular set or collection.

For correctness and clarity, we exclusively use the term *multiplicity* when talking about modeling edge labels within a graph schema. Let's dig a little deeper into the two options available when you apply the definition of multiplicity in your graph model.

#### Modeling multiplicity in the GSL

The application of multiplicity to your graph's schema comes down to understanding the different kinds of groups of adjacent vertices that are possible. There are only two: a set or a collection.

*Set*
: A set is an abstract data type that stores unique values.

*Collection*
: A collection is an abstract data type that stores nonunique values.

In a set of adjacent vertices, there can be only one instance of a vertex. In a collection of adjacent vertices, there could be many instances of a vertex. We illustrate the difference between these concepts in Figure 2-12.

2 James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, vol. 2 (Reading, MA: Addison-Wesley, 1999).

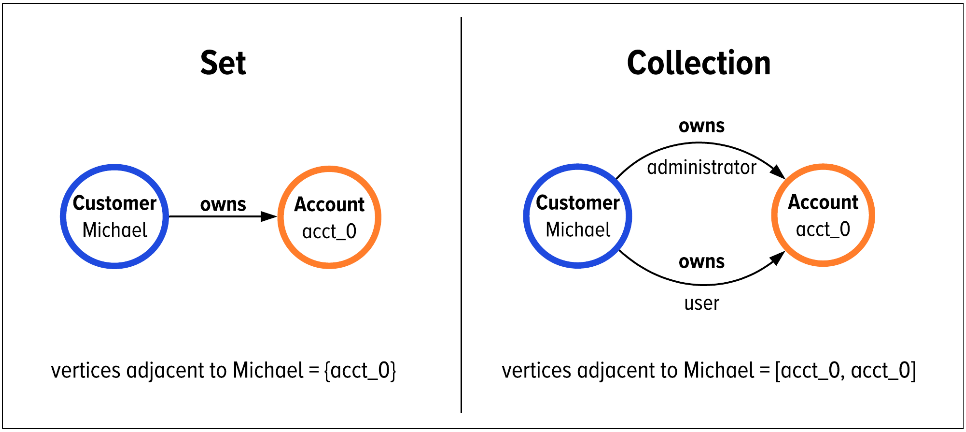



*Figure 2-12. The two options for applying multiplicity to the group of adjacent vertices for a specific edge label*

The graph on the left in Figure 2-12 shows that the group of vertices adjacent to `Michael` is a set: `{acct_0}`. This means that we want to have at most one edge between a customer and an account in our database. The graph on the right in Figure 2-12 shows that the group of vertices adjacent to `Michael` is a collection: `[acct_0, acct_0]`. This means that we want to have many edges between a customer and an account in our database. An example of when you would like many edges is when you want to represent that a customer is both administrator and user of an account.

> The most likely occasion for needing to decide about the multiplicity type is when you are modeling time on your edges. Do you want only the most recent edge in your database? Then you are thinking of your edge as a set. Do you want all of your edges over time? Then you are thinking of your edges as a collection. We will cover time on edges in Chapters 7, 9, and 12.

Let's look at how we would model the differences between the two graphs from Figure 2-12 in the GSL.

Figure 2-13 shows how we use a single line in the GSL to illustrate that we want to have at most one edge between two vertices.

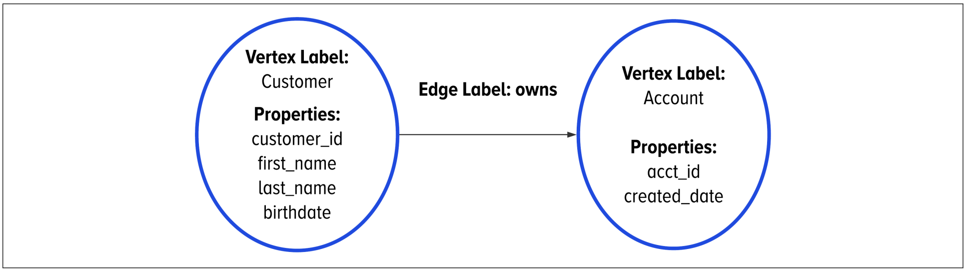



*Figure 2-13. In the GSL, we use a single line to indicate when the group of adjacent vertices needs to be a set*

In order to be able to model many edges between two vertices, we need a way for one edge to be different from another edge. In Figure 2-12, we added the `role` property to the edge so that each edge is different. Figure 2-14 shows how we use a double line and a property value in the GSL to illustrate that we want to have many edges between two vertices.

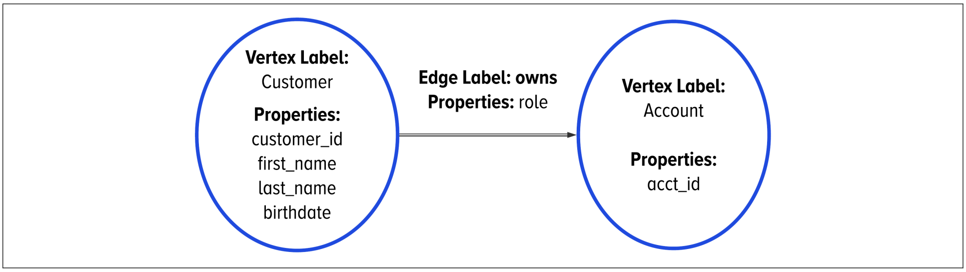



*Figure 2-14. In the GSL, we use a double line to indicate when the group of adjacent vertices needs to be a collection*

The trick to understanding multiplicity lies in understanding your data. If you need to have multiple edges between two vertices—because you need the group of connected vertices to be a collection rather than a set—then you need to define a property on the edge that makes it unique.

## Full Example Graph Model

Using GSL, the data from Table 2-1 translates into the conceptual graph model shown in Figure 2-15.

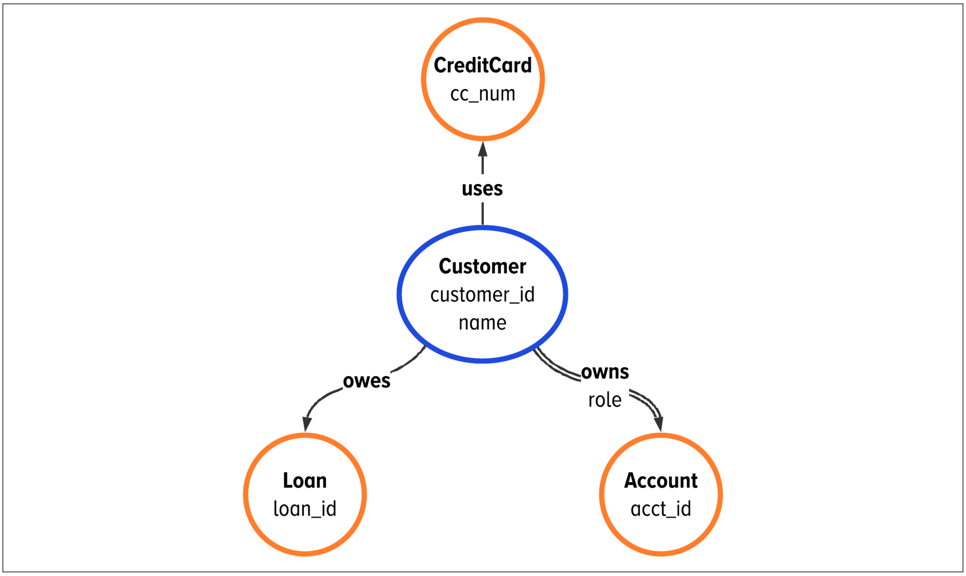



*Figure 2-15. The starting point for our first example: a basic model for customer data from financial services*

We refer to the image in Figure 2-15 as a *conceptual graph model*. This model creates your graph database schema. This conceptual graph model shows a customer and three different pieces of data related to the customer. These four entities translate to four separate vertex labels: `Customer`, `Account`, `Loan`, and `CreditCard`.

These four pieces of data are related in three ways: customers own accounts, use credit cards, and owe loans. This creates three edge labels in the conceptual graph model: `owns`, `uses`, and `owes`, respectively. All three edge labels are directed; there are no bidirectional edge labels in this example. Further, we see that the edge labels `uses` and `owes` will have at most one edge between two vertices, whereas `owns` can have many edges.

The final piece of Figure 2-15 to explore is the properties shown on each vertex label. These are the properties that we can find in the data from Table 2-1. A `Customer` has two properties: `customer_id` and `name`. The `Account`, `CreditCard`, and `Loan` vertex labels each have only one property: `acct_id`, `cc_num`, and `loan_id`, respectively. In each case, the property is the unique identifier for the data.

It is important to understand the difference between Figure 2-15 and the instance data we showed in Figure 2-4. Figure 2-15 shows the conceptual graph model for your database schema using GSL. Figure 2-4 shows what the data will look like in your graph database.

# Relational Versus Graph: Decisions to Consider

The most difficult evaluations of relational versus graph technologies are those that intertwine techniques for database modeling with those of data analysis. We want to conclude with a few notes on each of those topics to set you up for more effective evaluation processes.

## Data Modeling

Graph data modeling is very similar to relational data modeling; the main difference is in the consideration of relationships between entities. Graph technology is optimized for relationship-first data organization so as to provide direct access to an entity's relationships in the database. Given this, you will want to explore graph technology if the relationships between your entities are the most important features of your data.

In contrast to relational technology, graph technology was created to minimize the transition from mental model to data storage and retrieval. With graph technologies, the conceptual data model is the actual physical data model. That is, you don't have to specifically do any physical data modeling, as the graph database optimizes the storage and physical layout based on the logical model alone. This is achieved by storing the edges for a vertex in structures that give direct access to the edges associated to a vertex.

In our experience, the shorter translation from mental model to data storage is one of the primary reasons architects are turning from relational to graph technologies. When using graph technology, you can draw one image that represents both the conceptual understanding and the physical organization of your data. This shorter interpretation from conceptual to physical data organization creates a more powerful way to envision, discuss, and apply the relationships within your data. Without graph thinking and technology, this was previously unachievable.

## Understanding Graph Data

Applied graph theory empowers the appeal of using graph technology in your application. Graph technology gives you the means to understand both *if* and *how well* your data is connected. Specifically, concepts such as neighborhoods and degree open up to a new understanding about your data that is not possible with relational technologies.

The nuances between the worlds of graph schema and graph data are very important. Introducing graph technology to your team comes with a learning curve of new terms, concepts, and applications. One of the most effective ways to keep yourselves from being blocked is to understand which concepts apply to database modeling and which apply to data analysis at the application level.

## Mixing Database Design with Application Purpose

We have learned from our experiences that teams often confuse concepts about graph data analysis and graph schema. As we see it, interchanging terms from graph schema and graph data is the same as confusing the following two concepts: pie charts and foreign key constraints.

Let's unpack that.

Relational technologies are great for setting up a database that creates reports and summaries of data, like pie charts. Something like a pie chart visualizes a metric about the data. The application (the pie chart) is a completely different, and unrelated, concept from relational schema design, like selecting foreign key constraints between tables.

An application of your relational database is to create pie charts, and the database's schema requires designing with foreign keys to make it possible.

When getting started with graph technology, this same distinction applies. After you have set up your graph database, you use it to understand the connectivity of your data. Specifically, you can find the distance between two vertices in your graph. This is at the application level and uses data to understand the connections within the data. This is made possible by creating a graph database schema with vertex labels, edge labels, and properties.

An application of your graph database is to calculate the distance between vertices, and the database's schema requires designing with edge labels and vertex labels to make that possible.

The important takeaway here is to understand the differences between creating database schema and analyzing graph data.

Up until now, the flood of graph thinking has introduced many waves of terminology and complexity. In this chapter, we hope to have clearly delineated the techniques for creating database models as well as a few for analyzing graph data.

# Summary

This chapter set out to translate the concepts and terms that are used across multiple communities. Our goals for this content were to provide background and information for the following three objectives:

1. Is graph technology better for my problem than relational technology?
2. How do I think about my data as a graph?
3. How do I model a graph's schema?

In our experience, these three questions are the primary topics of conversation within development teams that are evaluating graph technology for their application stack.

We selected the content in this chapter as the minimum topic set needed to answer these questions. The terms and topics in this chapter represent the starting point for understanding data modeling, graph data, and application design in relational or graph systems. Combined with the GSL, the foundational concepts throughout this chapter represent what you need to know to get started with graph technology. At this point, you are equipped with the terminology and concepts you need to begin your first application design and evaluation.

We admittedly haven't given you much that answers our first question. This is because we can't answer it directly for you. Your team's need for graph technology for your application comes down to the applicability of the concepts and terms presented throughout this chapter. To boil it down: if relationships matter to your data, then graph technology will be the right answer for your team. Only you can determine that about your data.

On the other hand, we can help you navigate the use of relational or graph technology for specific use cases. The next chapter will walk you through a common starting use case in which teams typically put relational versus graph technologies to the test. Without further ado, let's start with the foundation that companies have successfully built as the gateway to using graph data in your business: the single view of your customer.

# CHAPTER 3
# Getting Started: A Simple Customer 360

We have often seen technical teams understand the benefits of graph thinking in the context of discussing a data problem that most large businesses face: trying to extract value across disparate data sources. Standing at a whiteboard sketching out the problem inevitably produces one hairy graph.

You can imagine this same scenario. You are drawing at a whiteboard and actively discussing how your system's data is spread in different silos across the company's systems. Your team agrees that what it really needs is direct access to your customers and their data. To illustrate this, almost every time, your coworker draws the customer at the center of the whiteboard and connects the relevant data to the customer. After stepping back, you all realize your colleague just drew a graph.

In our experience, these whiteboarding exercises illustrate the power of using graph thinking to build a data management solution. Graph applications start with data management because, either conceptually or physically, previous technology choices forced us to shape graph data into tabular solutions. The problem is that tabular-shaped data is no longer a one-size-fits-all design for today's applications.

This is especially true for those applications that have to cater to the user's demand for personalized context. The rising demand for personalization has put top-down pressure on data availability and relevancy. This pressure has forced organizations to integrate disparate data and ensure that data ties users to their digital experience.

When teams huddle around the drawing board to re-architect their systems to deliver personalization, they encounter a new problem. How does a single system unify data, function in real time, and relate the data back to the end user? Existing relational tools are great for those parts of the process that require the data to fit well in a row-and-column format.

However, relational tools are not well suited for delivering certain shapes of data—specifically, deeply connected data.

At this point in the whiteboarding session, we have reached a significant discussion topic: identifying and comparing solutions. The solution design process often introduces multiple technologies. The subsequent debate around which technology to choose can be divisive and never-ending.

## Chapter Preview: Relational Versus Graph

To address this common pressure point, the main goals of this chapter are as follows:

1. Define and formalize a common starting application for graph data
2. Build out an example application architecture with relational and graph technologies
3. Provide a guide for making the right choice for your system's needs

Throughout the rest of this chapter, we are going to introduce and motivate the use case that we just described in the whiteboard story. Afterwards, we are going to step through the implementation details of this type of application, starting with a relational system. Then we will follow the same process with a graph system. We will close this chapter with a discussion of how to select which technology is best for your application. Getting this right will help you find roots in the seemingly circular and never-ending debate over when, where, and how to apply graph technology to resolve your data management needs.

## The Foundational Use Case for Graph Data: C360

As we illustrated with the whiteboard story, tech teams all over the world are realizing the utility of graph data to solve their data management problems. For this type of problem, the difference between old and new solutions lies in the usefulness of modeling, storing, and retrieving relationships within your data.

Applications that aim to focus on the relationships in their data have the initial challenge of transforming and unifying data across relational systems. This transformation requires us to reorganize our thinking and processes from organizing entities to organizing relationships. Similar to the whiteboard drawing from earlier, the new approach to organizing your data according to its relationships is usually very close to what we have in Figure 3-1.

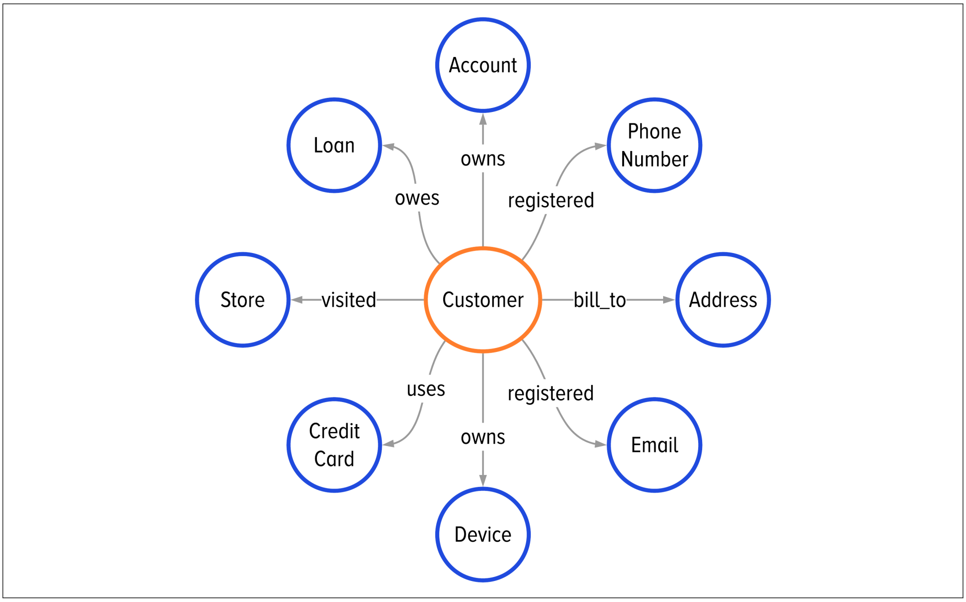



*Figure 3-1. An exercise in graph thinking to yield a conceptual graph model*

Adopters of graph technologies independently converged on naming this type of solution a *Customer 360* application, commonly abbreviated C360. The vision of a C360 project, like what is illustrated in Figure 3-1, is to engineer an application around the relationships between the important entities in your business.

You can envision the goal of a C360 application; there is a central object, your customer, and the customer's relationships to other integral pieces of data. These pieces of data are likely those that are most relevant to your business's domain. Commonly, we see teams start with the customer's family, methods of payment, or important identification details. This particular application within financial services is designed to answer the following types of questions about your customer:

1. Which credit cards does this customer use?
2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

The idea to unify consumer data into a single application is not new. Existing solutions such as data warehouses or data lakes provide single systems in which consumer data is stored. The problem here isn't in the integration of a business's data but in its accessibility. The era of graph thinking made us revisit these solutions in search

of a way to make this data more available and representative of the individual's experience.

Think of it this way: would you rather spend the day fishing, or would you like to get quick access to your dinner?

The difference between fishing for or ordering your dinner is similar to putting your data in a data lake or organizing your data for quick retrieval. The demands of today's digital applications require architects to focus on the quick delivery of data. Graph technologies allow architects to build deeply connected retrieval systems to complement longer search expeditions across data lakes.

## Why Do Businesses Care About C360?

Consumers interact with your company in an omnichannel fashion; they seamlessly transition from your mobile or web applications to your social media feeds and physical storefronts. Across all of these channels, they are experiencing an integrated perception of your brand. Companies that match this expectation by creating a unified digital experience are seeing revenue increases of up to 10%. These revenue increases are measured to be two to three times faster than those of companies that have not unified their customers' digital experiences.[1]

The secret ingredient behind this observed revenue growth is an application that unifies all customer data. Bringing together all of your customers' data into an application mirrors each customer's experience with your brand. In other words, it is a C360 application.

There are myriad creative examples of early innovators who have deployed interesting C360 applications. One of the more unique examples comes from Baidu (the Google of China) and Kentucky Fried Chicken. Through a unified data platform, Baidu teamed up with KFC to deliver order recommendations. Their collaborative solution identifies customers, accesses their order history, and returns order recommendations. This integration of data across these two industries has proven to be a unique and profitable example of C360 technologies.

A C360 application is the starting place for implementing graph thinking in your business. Getting this right provides a solid foundation for introducing graph data into your system's architecture. We have found that one of the most common mistakes made by architects and system designers is to move too quickly from the conceptual model to implementation details with graph technologies. There is more to consider here, and we want to use our experience to guide you through your own evaluation throughout the rest of this chapter.

---

1 Mark Abraham, et al. "Profiting from Personalization." Boston Consulting Group, May 8, 2017. *https://www.bcg.com/publications/2017/retail-marketing-sales-profiting-personalization.aspx*.

## Implementing a C360 Application in a Relational System

The goal of this section is to briefly introduce how to build out a relational system to store the C360 data. This section does not serve as a complete introduction to this class of system architecture. Rather, our goal is to introduce the minimum needed to understand the complexities of using a relational system for a C360 application.

To illustrate the process from data modeling through queries, we will be using the same data introduced in Table 2-1. For convenience, we are sharing the table of data again in this chapter—see Table 3-1. For a complete refresher on the generation, meaning, and details of this data, refer to the discussion in "Data for Our Running Example" on page 23.

*Table 3-1. A sample of the data created to illustrate the technology choices in this chapter*

| customer_id | name | acct_id | loan_id | cc_num |
|---|---|---|---|---|
| customer_0 | Michael | acct_14 | loan_32 | cc_17 |
| customer_1 | Maria | acct_14 | none | none |
| customer_2 | Rashika | acct_5 | none | cc_32 |
| customer_3 | Jamie | acct_0 | loan_18 | none |
| customer_4 | Aaliyah | acct_0 | [loan_18, loan_80] | none |

The two technologies we will be using to illustrate a relational implementation are SQL and Postgres. SQL, which stands for Structured Query Language, is the programming language used to communicate with a relational database. We have chosen to use the Postgres RDBMS because of its wide applicability and origins within the open source community.

### Data Models

After agreeing on a conceptual model, like that shown in Figure 2-1, you can move on to the design of your relational database. Traditionally, you would create an *entity-relationship diagram*, or ERD. An ERD is a logical representation of your data model and is a typical starting point for a relational database design.

In Figure 3-2, each square represents an entity that will become a table in the relational database. The *attributes*, or descriptive properties about each entity, are listed within each square. As already seen in the data, each entity will have a unique identifier. A customer will be uniquely identified by its `customer_id`, an account by its `acct_id`, and so on. Customers also have names and, in larger applications, other attributes.

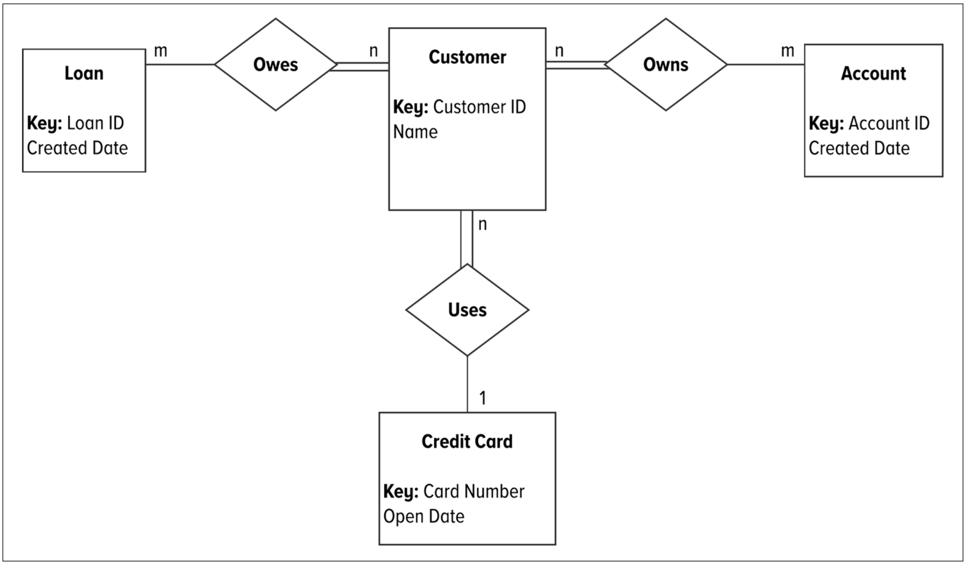



*Figure 3-2. An entity-relationship diagram for a relational implementation of this C360 application*

The diamond shapes between entities in Figure 3-2 represent the connection from one entity to another. The *cardinality* of the connection is indicated above and below or to the left and right of the diamond shape. In this data, we have two types of connections: one-to-many and many-to-many.

Let's start with the *one-to-many* connection that is shown between customers and credit cards. In this example, a customer can have many credit cards, but a credit card can only have one customer. This one-to-many connection describes the *cardinality* between customers and credit cards and is illustrated with the `n to 1` connection between customers and credit cards in Figure 3-2.

The other type of connection we see in our data is a *many-to-many* connection. There are two many-to-many connections in our data: customers to accounts and customers to loans. We know from our data that a customer can have many accounts, and one account can have many customers. The same is true for loans. We say that customers to loans is a many-to-many connection and illustrate this in Figure 3-2 with the `n to m` notation on the connection.

Before creating tables and inserting data, we need to translate our logical data model into a physical data model. Specifically, we need to translate the entities and connections from the ERD illustrated in Figure 3-2 into tables with primary and foreign keys.

We need two types of keys for this implementation: primary and foreign keys. A *primary key* is a uniquely identifying piece of data, such as a customer's ID or credit card number, that we will use to access the information in its table. A *foreign key* is a uniquely identifying piece of data that we will use to access the information in a different table, such as storing a customer's ID alongside their credit card information. We store a customer's ID with the customer's credit card information so that we can use it to look up all of their information in a different table, namely in the customer table.

Let's take a look at how the keys and data from Figure 3-2 map into the physical data model shown in Figure 3-3.

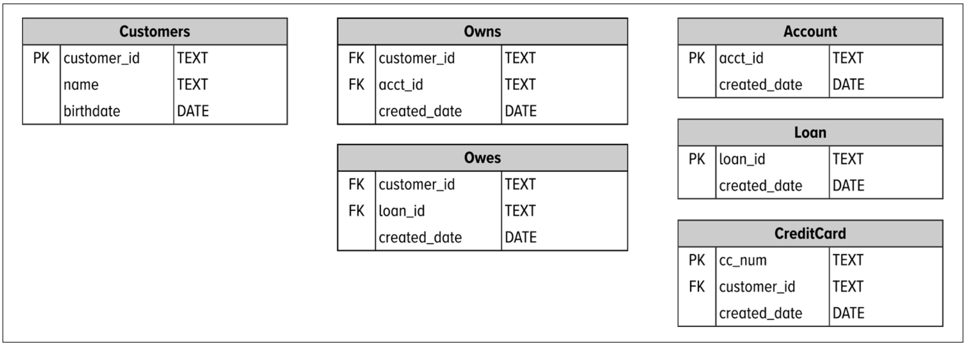



*Figure 3-3. A physical data model for a relational implementation of this C360 application*

We expected to see at least four tables in Figure 3-3—one table for each entity. Specifically, Figure 3-3 has one table per entity type: customer, account, loan, and credit card. For each of those tables, we see additional attributes that describe the entity. The most important attribute for each of these entities is its primary key. Each primary key is indicated with a `PK` next to the row. The primary keys we have for each table are the `customer_id`, `acct_id`, `loan_id`, and `cc_num`, respectively. These are the unique identifiers that we will use to look up a specific row of information in the table.

Before we talk about the other two tables in Figure 3-3, let's examine the `CreditCard` table. This table has both a primary key and a foreign key. We are using a foreign key in this table to track the one-to-many relationship we created in our ERD. The `customer_id` is the foreign key (indicated with an `FK`) that will give us the ability to relate the credit card information back to a unique customer. Building a one-to-many relationship into your physical data model can be as easy as adding a foreign key to join you back to another entity table.

The last two tables to understand in our physical data model are the `Owns` and `Owes` tables. These tables are *join tables* that allow us to physically store the many-to-many connections in our data. The `Owns` table stores the many connections observed

between customers and the accounts that they own. The `Owes` table stores the many connections observed between customers and the loans that they owe. Since each customer can own an account only once and can owe a loan only once, the primary key of these join tables is a compound of the two foreign keys.

For example, the `Owns` table stores at least two pieces of information about each row in the table: the customer's unique identifier and the account's unique identifier. Given one row from this table, we can access both the customer's unique identifier to join back to the customer table and the account's unique identifier to join back to the account table. This join table is a common way to represent a many-to-many connection in a relational system.

## Relational Implementation

Given our physical data model, let's walk through creating the tables and inserting our sample data from Table 3-1 into the tables.

First, we want to create the customer table. The final data model for this is shown in Figure 3-4.

| Customers | | |
|---|---|---|
| PK | customer_id | TEXT |
| | name | TEXT |

*Figure 3-4. The customer table for the relational implementation*

The SQL statement for creating the customer table is:

```
CREATE TABLE Customers ( customer_id TEXT,
                         name TEXT,
                         PRIMARY KEY (customer_id));
```

Our data has five customers. Let's insert those five customers into this customer table:

```
INSERT INTO Customers (customer_id, name) VALUES
  ('customer_0', 'Michael'),
  ('customer_1', 'Maria'),
  ('customer_2', 'Rashika'),
  ('customer_3', 'Jamie'),
  ('customer_4', 'Aaliyah');
```

Our data in our relational database now has one table with five entries, as shown in Figure 3-5.

| Customers | |
|---|---|
| customer_0 | Michael |
| customer_1 | Maria |
| customer_2 | Rashika |
| customer_3 | Jamie |
| customer_4 | Aaliyah |

*Figure 3-5. The customer data within our relational database*

Next, let's add the other three entity tables for `Accounts`, `Loans`, and `CreditCards`. Their final data models are shown in Figure 3-6.

| Account | | |
|---|---|---|
| PK | acct_id | TEXT |
| | created_date | DATE |

| Loan | | |
|---|---|---|
| PK | loan_id | TEXT |
| | created_date | DATE |

| CreditCard | | |
|---|---|---|
| PK | cc_num | TEXT |
| FK | customer_id | TEXT |
| | created_date | DATE |

*Figure 3-6. The account, loan, and credit card tables for the relational implementation*

We will start by creating the two tables for `Accounts` and `Loans`:

```
CREATE TABLE Accounts ( acct_id TEXT,
                        created_date DATE DEFAULT CURRENT_DATE,
                        PRIMARY KEY (acct_id));

CREATE TABLE Loans ( loan_id TEXT,
                     created_date DATE DEFAULT CURRENT_DATE,
                     PRIMARY KEY (loan_id));
```

Next, let's insert the data for `Accounts` and `Loans`:

```
INSERT INTO Accounts (acct_id) VALUES
  ('acct_0'),
  ('acct_5'),
  ('acct_14');

INSERT INTO Loans (loan_id) VALUES
  ('loan_18'),
  ('loan_32'),
  ('loan_80');
```

At this point, we have one last entity table to create in our relational database: the table for `CreditCards`. Because credit cards have a one-to-many relationship with customers, we also need to insert the customer's ID as a foreign key. We create this table with:

```
CREATE TABLE CreditCards
    ( cc_num TEXT,
      customer_id TEXT NOT NULL,
      created_date DATE DEFAULT CURRENT_DATE,
      PRIMARY KEY (cc_num),
      FOREIGN KEY (customer_id) REFERENCES Customers(customer_id));
```

Looking back at the data from Table 3-1, we find each unique credit card and the customer who owns that card. From this information, we can create the following statements to insert this data into our relational database:

```
INSERT INTO CreditCards (cc_num, customer_id) VALUES
  ('cc_17', 'customer_0'),
  ('cc_32', 'customer_2');
```

Our relational database now has a total of four tables with data; see Figure 3-7.

| Customers | |
|---|---|
| customer_0 | Michael |
| customer_1 | Maria |
| customer_2 | Rashika |
| customer_3 | Jamie |
| customer_4 | Aaliyah |

| Accounts | |
|---|---|
| acct_0 | 2020-01-01 |
| acct_5 | 2020-01-01 |
| acct_14 | 2020-01-01 |

| CreditCards | | |
|---|---|---|
| cc_17 | customer_0 | 2020-01-01 |
| cc_32 | customer_2 | 2020-01-01 |

| Loans | |
|---|---|
| loan_18 | 2020-01-01 |
| loan_32 | 2020-01-01 |
| loan_80 | 2020-01-01 |

*Figure 3-7. The data in our relational database for the four entity tables*

The last two tables to create in our relational implementation are those for the many-to-many connections from `Customers` to `Accounts` and `Loans`. First, let's create the table that will join Customers to Accounts, as illustrated in Figure 3-8.

| Owns | | |
|---|---|---|
| FK | customer_id | TEXT |
| FK | acct_id | TEXT |
| | created_date | DATE |

*Figure 3-8. The join table from Customers to Accounts*

In SQL, we create this table with:

```
CREATE TABLE Owns ( customer_id TEXT NOT NULL,
                    acct_id TEXT NOT NULL,
                    created_date DATE DEFAULT CURRENT_DATE,
                    PRIMARY KEY (customer_id, acct_id),
```

```
                    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
                    FOREIGN KEY (acct_id) REFERENCES Accounts(acct_id));
```

Remembering the data from Table 3-1, we find the following data to insert into the `Owns` table:

```
INSERT INTO Owns (customer_id, acct_id) VALUES
  ('customer_0', 'acct_14'),
  ('customer_1', 'acct_14'),
  ('customer_2', 'acct_5'),
  ('customer_3', 'acct_0'),
  ('customer_4', 'acct_0');
```

Now that we have some data in our `Owns` table (see Figure 3-9), we can see how to associate the data from a customer and an account in our data.

| Customers | |
|---|---|
| customer_0 | Michael |
| customer_1 | Maria |
| customer_2 | Rashika |
| customer_3 | Jamie |
| customer_4 | Aaliyah |

| Owns | |
|---|---|
| customer_0 | acct_14 |
| customer_1 | acct_14 |
| customer_2 | acct_5 |
| customer_3 | acct_0 |
| customer_4 | acct_0 |

| Accounts | |
|---|---|
| acct_0 | 2020-01-01 |
| acct_5 | 2020-01-01 |
| acct_14 | 2020-01-01 |

*Figure 3-9. The customer, account, and join table data*

The final step in creating our relational database is to create the `Owes` table to associate a customer to their loan, and vice versa. The final data model for this join table is shown in Figure 3-10.

| Owes | | |
|---|---|---|
| FK | customer_id | TEXT |
| FK | loan_id | TEXT |
| | created_date | DATE |

*Figure 3-10. The join table from Customers to Loans*

In SQL, we create this table in our relational database via:

```
CREATE TABLE Owes ( customer_id TEXT NOT NULL,
                    loan_id TEXT NOT NULL,
                    created_date DATE DEFAULT CURRENT_DATE,
                    PRIMARY KEY (customer_id, loan_id),
                    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
                    FOREIGN KEY (loan_id) REFERENCES Loans(loan_id));
```

Finally, we can extract one last connection from the data in Table 3-1 and insert all observations of customers who owe loans into the `Owes` table:

```
INSERT INTO Owes (customer_id, loan_id) VALUES
  ('customer_0', 'loan_32'),
  ('customer_3', 'loan_18'),
  ('customer_4', 'loan_18'),
  ('customer_4', 'loan_80');
```

The full picture of our data in our relational database is shown in Figure 3-11.

| Customers | |
|---|---|
| customer_0 | Michael |
| customer_1 | Maria |
| customer_2 | Rashika |
| customer_3 | Jamie |
| customer_4 | Aaliyah |

| Owns | |
|---|---|
| customer_0 | acct_14 |
| customer_1 | acct_14 |
| customer_2 | acct_5 |
| customer_3 | acct_0 |
| customer_4 | acct_0 |

| Owes | |
|---|---|
| customer_0 | loan32 |
| customer_3 | loan18 |
| customer_4 | loan18 |
| customer_4 | loan80 |

| Accounts | |
|---|---|
| acct_0 | 2020-01-01 |
| acct_5 | 2020-01-01 |
| acct_14 | 2020-01-01 |

| Loans | |
|---|---|
| loan18 | 2020-01-01 |
| loan32 | 2020-01-01 |
| loan80 | 2020-01-01 |

| CreditCards | | |
|---|---|---|
| cc_17 | customer_0 | 2020-01-01 |
| cc_32 | customer_2 | 2020-01-01 |

*Figure 3-11. The complete mapping of the data into a relational database*

## Example C360 Queries

Now that the data is in our relational database, we need to ask the four fundamental queries for our C360 application:

1. Which credit cards does this customer use?
2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

For our relational system, we are asking these four questions in a specific order for two reasons. First, we want to start slowly with a natural progression toward asking more detailed questions about a person from a database. Second, we structured these questions so that the technical implementation builds upon each statement to conclude with the final SQL statement.

### Query: Which credit cards does this customer use?

First, let's use our relational database to query for the credit cards owned by `customer_0`. The data for this query is directly available from the `CreditCards` table. If we just want the credit card information, we can query the table with the following SQL query:

```
SELECT * from CreditCards WHERE customer_id = 'customer_0';
```

This query will return the following data:

| cc_num | customer_id | created_date |
|---|---|---|
| cc_17 | customer_0 | 2020-01-01 |

It is likely that you really wanted to view the customer's data alongside their credit card information. This requires us to join the `Customers` table with the `CreditCards` table. In SQL, this would be done via:

```
SELECT Customers.customer_id,
       Customers.name,
       CreditCards.cc_num,
       CreditCards.created_date
FROM Customers
LEFT JOIN CreditCards ON (Customers.customer_id = CreditCards.customer_id)
WHERE Customers.customer_id = 'customer_0';
```

This query will return the following data:

| Customers.customer_id | Customers.name | CreditCards.cc_num | CreditCards.created_date |
|---|---|---|---|
| customer_0 | Michael | cc_17 | 2020-01-01 |

Getting access to a customer alongside their credit card information requires only one join statement because of the one-to-many relationship between customers and credit cards. When we need to look at the data about customers and their accounts, things get a little tricky.

### Query: Which accounts does this customer own?

Next, let's query the relational database to answer the question: which accounts does `customer_0` own? For this query, we will need to use the join table `Owns` to join together the customer table with the account table. The SQL query for this is:

```
SELECT Customers.customer_id,
       Customers.name,
       Accounts.acct_id,
       Accounts.created_date
FROM Customers
LEFT JOIN Owns ON (Customers.customer_id = Owns.customer_id)
```

```
LEFT JOIN Accounts ON (Accounts.acct_id = Owns.acct_id)
WHERE Customers.customer_id = 'customer_0';
```

This query starts by accessing the data about `customer_0` from the customer table. Next, we find all foreign key pairs from the `Owns` table that have a matching `customer_id`. There is only one entry in the `Owns` table for this customer because `customer_0` owns only one account. From here, we follow the foreign keys of the accounts over to the `Accounts` table to extract the account information. The resulting data looks like:

| Customers.customer_id | Customers.name | Accounts.acct_id | Accounts.created_date |
|---|---|---|---|
| customer_0 | Michael | acct_14 | 2020-01-01 |

### Query: Which loans does this customer owe?

The next question uses the same structure but traces from the customer table to the `Loans` table by using the `Owes` join table. This question is asking for the customer's information alongside their loan details. For this query, let's use the data about `customer_4`. The SQL statement for this query is:

```
SELECT Customers.customer_id,
       Customers.name,
       Loans.loan_id,
       Loans.created_date
FROM Customers
LEFT JOIN Owes ON (Customers.customer_id = Owes.customer_id)
LEFT JOIN Loans ON (Loans.loan_id = Owes.loan_id)
WHERE Customers.customer_id = 'customer_4';
```

The resulting data is:

| Customers.customer_id | Customers.name | Loans.loan_id | Loans.loan_id |
|---|---|---|---|
| customer_4 | Aaliyah | loan_18 | 2020-01-01 |
| customer_4 | Aaliyah | loan_80 | 2020-01-01 |

### Query: What do we know about this customer?

Each of these queries is building up the individual pieces required to ask the main query for a C360 application: for a specific customer, tell me everything we know about them. This query brings together each of the three previous queries into one statement. The following SQL statement uses all six tables across our relational database to find all information about one customer. Let's use `customer_0` again in this final example:

```
SELECT Customers.customer_id,
       Customers.name,
       Accounts.acct_id,
```

```
        Accounts.created_date,
        Loans.loan_id,
        Loans.created_date,
        CreditCards.cc_num,
        CreditCards.created_date
FROM Customers
LEFT JOIN Owns ON (Customers.customer_id = Owns.customer_id)
LEFT JOIN Accounts ON (Accounts.acct_id = Owns.acct_id)
LEFT JOIN Owes ON (Customers.customer_id = Owes.customer_id)
LEFT JOIN Loans ON (Loans.loan_id = Owes.loan_id)
LEFT JOIN CreditCards ON (Customers.customer_id = CreditCards.customer_id)
WHERE Customers.customer_id = 'customer_0';
```

This will transform the data about `customer_0` from across the database into the following:

| customer_id | name | acct_id | created_date | loan_id | created_date | cc_num | created_date |
|---|---|---|---|---|---|---|---|
| customer_0 | Michael | acct_14 | 2020-01-01 | loan_32 | 2020-01-01 | cc_17 | 2020-01-01 |

The four questions we demonstrated in this section are just scratching the surface of the SQL query language. And we addressed only the fundamentals of SQL: `SELECT-FROM-WHERE` with basic joins. Even though our questions can be stated very simply, the required queries become increasingly complex. It is even harder to follow the data throughout this system to understand which data is related to which customer.

## Implementing a C360 Application in a Graph System

Now that we have walked through a relational implementation, let's dig into transforming our sample data into a graph database implementation. Let's revisit the conceptual model, shown in Figure 3-12, before we dig into the implementation details in this section.

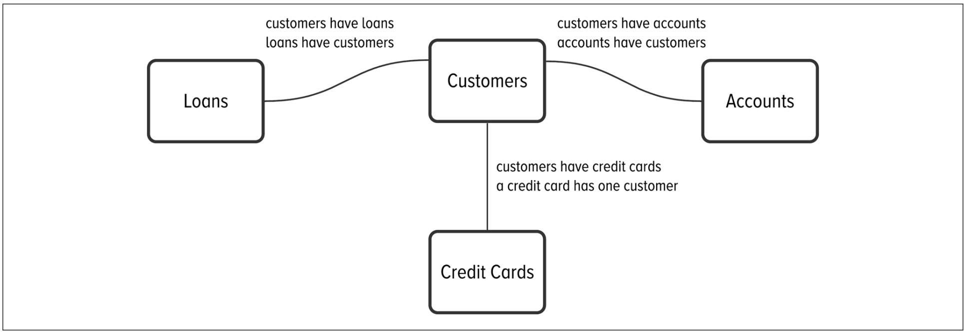



*Figure 3-12. A conceptual description of the relationships observed in the data in Table 3-1*

For this example, we are going to use the Gremlin query language—the most widely implemented graph query language— and DataStax Graph schema APIs. We are choosing to use Gremlin due to its wide adoption across the graph database community and its roots in open source. Our overarching objective in this book is to build up to implementing graphs within a distributed, partitioned environment. Given this goal, we will be using the DataStax Graph schema APIs to build up to working with distributed graphs.

## Data Models

Compared to relational models, there is a smaller transition from a conceptual model to a graph data model. This lower bar illustrates the power of a database implementation that more closely represents your natural way of expressing data.

Using the GSL from "The Graph Schema Language" on page 33, Figure 3-13 contains a property graph model for our example data. The first benefit to notice is the shorter transition from conceptual (Figure 3-12) to logical data modeling for graph implementations.

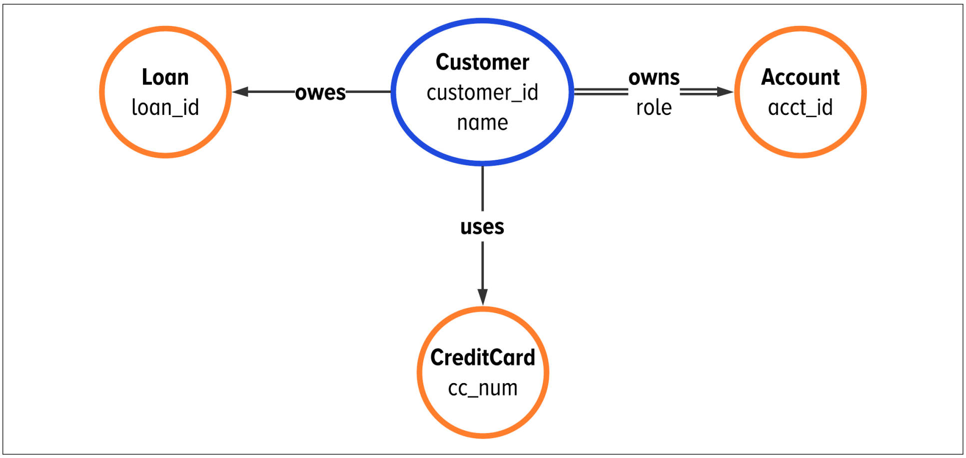



*Figure 3-13. A data model for a graph-based implementation of this C360 application*

There are four vertex labels in Figure 3-13: `Customer`, `Account`, `CreditCard`, and `Loan`. These vertex labels are shown in bold on the entities in the data model. There are three edge labels in Figure 3-13: `owns`, `uses`, and `owes`. These edge labels are shown in bold on the relationships in the data model.

Last, we find a few places in Figure 3-13 where we are using properties. A `Customer` vertex will have two properties: `customer_id` and `name`. You can see the properties for each vertex listed underneath the vertex labels. And we have also included a `role` on the `owns` edge label.

## Graph Implementation

With graph databases, our first implementation step will be to create the graph so that we can add the graph's schema. Once we have set up the schema, we will be ready to insert data into the database.

The code for creating a graph is as follows:

```
system.graph("simple_c360").create()
```

We took care of installing and setting up the graphs for you in the provided technical assets. If you want to dig into those steps on your own, you can find the step-by-step instructions in the DataStax Docs. We will not be covering those topics in this book.

Let's dive straight into creating graph schema. If you would like, you can follow along in the DataStax Studio Notebook we created for this chapter, `Ch3_SimpleC360`. DataStax Studio gives you a notebook environment for developing with DataStax products and is the best way to implement this book's examples. The notebooks are available in our book's GitHub repository.

### Creating your graph's schema

First, let's create the `customer` vertex label. Our customer data has a unique ID and a name:

```
schema.vertexLabel("Customer").
       ifNotExists().
       partitionBy("customer_id", Text).
       property("name", Text).
       create();
```

Let's finish the vertex label creation by adding the vertex labels for accounts, loans, and credit cards:

```
schema.vertexLabel("Account").
       ifNotExists().
       partitionBy("acct_id", Text).
       create();

schema.vertexLabel("Loan").
       ifNotExists().
       partitionBy("loan_id", Text).
       create();

schema.vertexLabel("CreditCard").
       ifNotExists().
       partitionBy("cc_num", Text).
       create();
```

At this point, we have four tables in the database—one table per vertex label. The last step is to add the relationships from the customer to each of the other entities in the data model.

For this example, we selected to model the edges coming out of the customer vertex and into the other vertex types. These edges have direction; it comes from the customer and goes to `Accounts`, `Loans`, and `CreditCards`. When we create an edge label, this direction matters. Let's look at an example for creating the `owes` relationship between a customer and their account:

```
schema.edgeLabel("owes").
       ifNotExists().
       from("Customer").
       to("Loan").
       create();
```

The direction of this edge label is set with the `from` and `to` steps. The edge comes from the vertex label `Customer` and goes to the `Loan` vertex label.

There are two more edge labels to create: one from the customer to their credit card and another from the customer to their account. The `owns` edges will also have the `role` property stored on the edge:

```
schema.edgeLabel("uses").
       ifNotExists().
       from("Customer").
       to("CreditCard").
       create();

schema.edgeLabel("owns").
       ifNotExists().
       from("Customer").
       to("Account").
       property("role", Text).
       create();
```

We read the label to say the edge `owns` is coming *from* the customer and going *to* the account and has a property called `role`.

#### Inserting your graph data

With our graph schema in place, we can add our sample data into this graph database. We will start by adding one piece of data—the vertex for `Michael`:

```
michael = g.addV("Customer").
            property("customer_id", "customer_0").
            property("name", "Michael").
            next();
```

When adding vertices into your graph, the `addV` step requires you to provide the full primary key. Otherwise, you will see an error like the one shown in Figure 3-14.

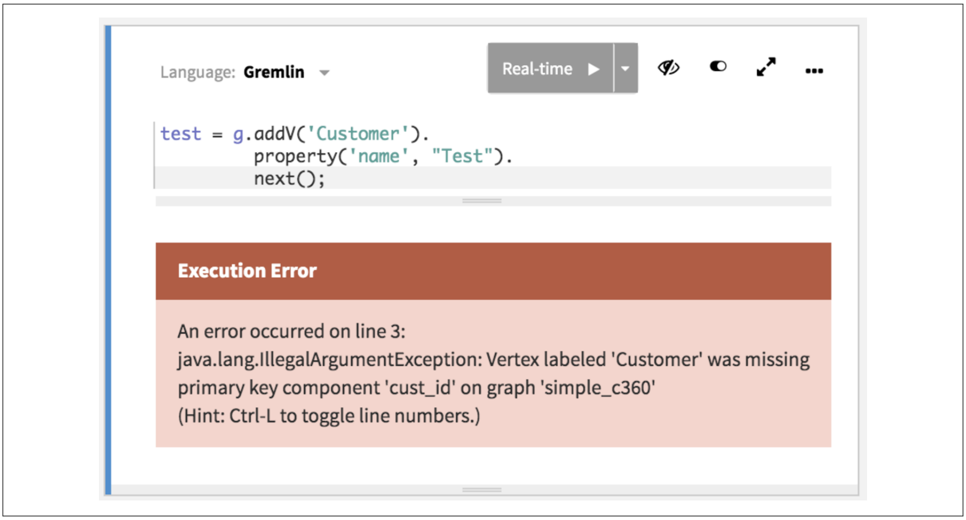



*Figure 3-14. An example of an error you may experience if you forget to include the full primary key when inserting a new vertex*

Next, let's add the vertices for Michael's account, loan, and credit card:

```
acct_14 = g.addV("Account").
            property("acct_id", "acct_14").
            next();

loan_32 = g.addV("Loan").
            property("loan_id", "loan_32").
            next();

cc_17 = g.addV("CreditCard").
          property("cc_num", "cc_17").
          next();
```

The `next()` step is a terminal step in Gremlin. It returns the first result from the end of a traversal. In the preceding example, we are returning the vertex object that we just added into the graph and storing it into an in-memory variable.

Now, we have four disconnected pieces of data in our graph database. As before, we stored each vertex object in variables called `acct_14`, `loan_32`, and `cc_17` to be used later. The database essentially has four vertices with no edges, as seen in Figure 3-15.

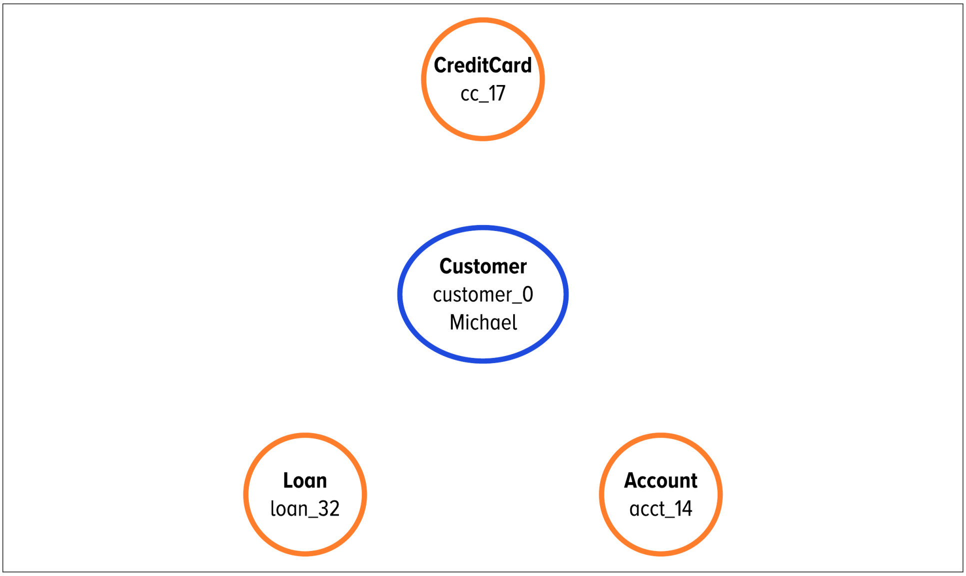



*Figure 3-15. The data currently in our graph database*

Let's introduce some connectivity between the data. To do so, we need to add three edges from `customer_0` to the other vertices. Using the variables we just created, we can add an edge from the vertex `Michael` to the vertices `account`, `loan`, and `creditCard`, respectively:

```
g.addE("owns").
  from(michael).
  to(acct_14).
  property("role", "primary").
  next();

g.addE("owes").
  from(michael).
  to(loan_32).
  next();

g.addE("uses").
  from(michael).
  to(cc_17).
  next();
```

When adding edges into the database, we start by identifying the vertex from which the edge will be coming. In the preceding example, this is `Michael` because all edges will be starting *from* `Michael` and going *to* other pieces of data. These three edges create the first connected view of our data in our graph database, as shown in Figure 3-16.

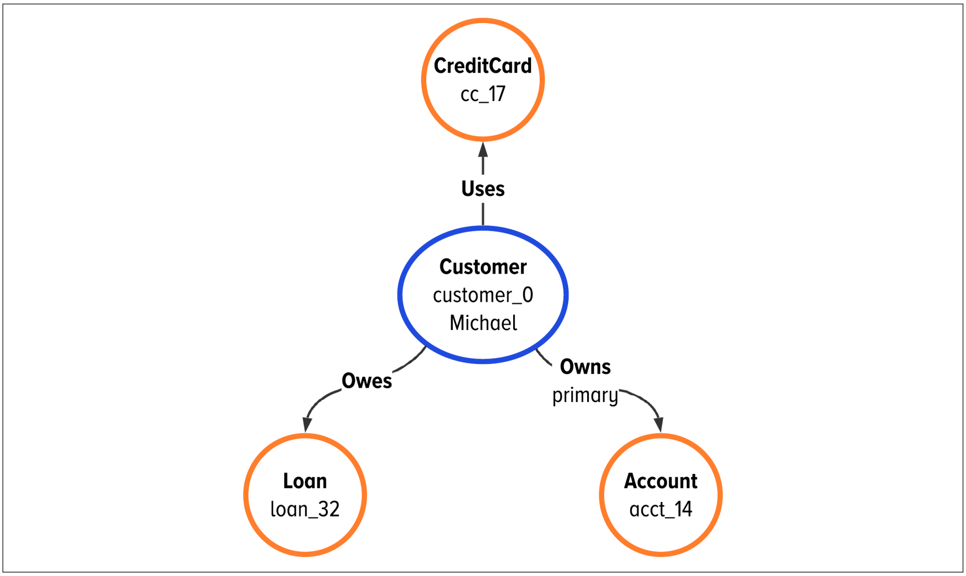



*Figure 3-16. A connected view of the data currently in our graph database*

From the example we already walked through, we know that `Maria` shares an account with `Michael`. Let's add the vertex for `Maria` and connect it to the account vertex we already created (see Figure 3-17):

```
maria = g.addV("Customer").
          property("customer_id", "customer_1").
          property("name", "Maria").
          next();

g.addE("owns").
  from(maria).
  to(acct_14).
  property("role", "limited").
  next();
```

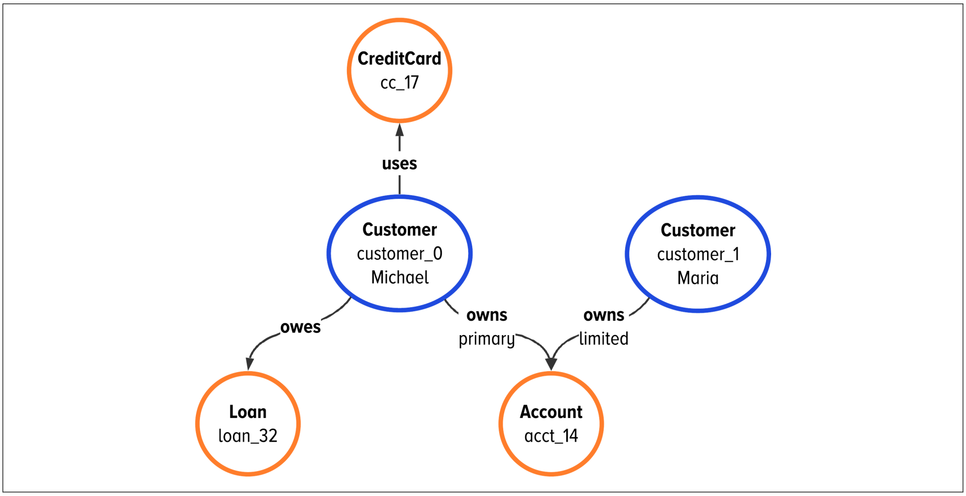



*Figure 3-17. The connected view of Michael's and Maria's data in our graph database*

Let's finish up this example by adding the vertices and edges about the remaining three customers:

```
// Data Insertion for Rashika
rashika = g.addV("Customer").
          property("customer_id", "customer_2").
          property("name", "Rashika").
          next();
acct_5 = g.addV("Account").
           property("acct_id", "acct_5").
           next();
cc_32 = g.addV("CreditCard").
          property("cc_num", "cc_32").
          next();
g.addE("owns").
  from(rashika).
  to(acct_5).
  property("role", "primary").
  next();
g.addE("uses").
  from(rashika).
  to(cc_32).
  next();

// Data Insertion for Jamie
jamie = g.addV("Customer").
          property("customer_id", "customer_3").
          property("name", "Jamie").
          next();
acct_0 = g.addV("Account").
           property("acct_id", "acct_0").
           next();
```

```
loan_18 = g.addV("Loan").
          property("loan_id", "loan_18").
          next();
g.addE("owns").
  from(jamie).
  to(acct_0).
  property("role", "primary").
  next();
g.addE("owes").
  from(jamie).
  to(loan_18).
  next();

// Data Insertion for Aaliyah
aaliyah = g.addV("Customer").
         property("customer_id", "customer_4").
         property("name", "Aaliyah").
         next();
loan_80 = g.addV("Loan").
          property("loan_id", "loan_80").
          next();
g.addE("owns").
  from(aaliyah).
  to(acct_0).
  property("role", "primary").
  next();
g.addE("owes").
  from(aaliyah).
  to(loan_80).
  next();
g.addE("owes").
  from(aaliyah).
  to(loan_18).
  next();
```

These final statements complete the insertion of the sample data into our graph database. Figure 3-18 shows the final view of the data in the database.

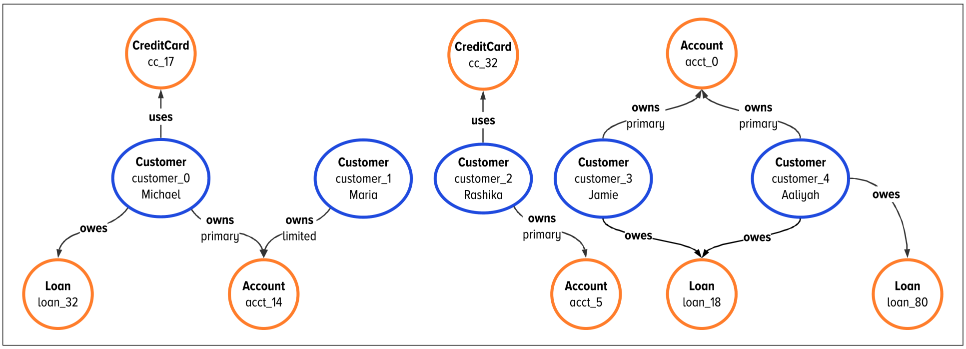



*Figure 3-18. The final view of the data in our graph database*

### Graph traversals

The Gremlin statements in this section are our first graph database queries. A graph database query is also called a *graph traversal*.

*Graph traversal*
: A graph traversal is an iterative process of visiting the vertices and edges of a graph in a well-defined order.

When using Gremlin, you start your traversals with a *traversal source*.

*Traversal source*
: A traversal source wraps two concepts together: the graph data you are traversing and traversal strategies, such as exploring data without indexes. The traversal sources you will use for examples in this book are `dev` (for development) and `g` (for production).

The queries in this section used the `g` traversal source. We will come back to using the `g` traversal source in Chapter 5 and in the production chapters.

For the rest of this chapter, we will be using the `dev` traversal source. We will always use the `dev` traversal source in this book when we are developing our graph traversals, like in this chapter, in Chapter 4, and in the development chapters. We use the `dev` traversal source because it allows us to explore our graph data without indexes on the data.

From here, let's move on to implementing the same queries as before, but with our graph data.

## Example C360 Queries

We like to think of querying a graph database, loosely, as the reverse of an SQL query. The common relational querying mindset is `SELECT-FROM-WHERE`. In graph, we are essentially asking the traversal to follow a similar pattern in reverse: `WHERE-JOIN-SELECT`.

You can think of a Gremlin query as beginning with `WHERE` you need to start from in your graph data. Then you tell the database to use relationships from your starting location to `JOIN` different pieces of data together. Last, you tell the database which data to `SELECT` and return. For a C360 application, our query loosely follows this `WHERE-JOIN-SELECT` pattern and is a great starting point for learning how to query a graph database.

With that in mind, let's revisit our C360 application queries, and then we will answer each question using the Gremlin query language and our graph database:

1. Which credit cards does this customer use?

2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

### Query: Which credit cards does this customer use?

First, let's use our graph database to query for the credit cards owned by `customer_0`. We can't just query for any credit card; we need to first access the vertex for `customer_0` and then walk to the credit card that is adjacent (connected) to `customer_0`. In Gremlin:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
        out("uses").                                  // JOIN
        values("cc_num")                              // SELECT
```

The language to the right of `//` in each line of code is an in-line comment to describe the logic happening in the code at left.

This query will return the following data:

```
"cc_17"
```

Let's break down this Gremlin query into the `WHERE-JOIN-SELECT` pattern. The first part of the query is `dev.V().has("Customer", "customer_id", "customer_0")`. We say that this step is finding *where* you are starting your graph traversal; we are starting by finding a vertex with label `Customer` that has `customer_id` equal to `customer_0`. The second step in this traversal is `out("uses")`. This step is *joining* the customer to their credit card data. The last step is to pick the data you want to return. This is the `values("cc_num")` step. This part of the Gremlin traversal is specifying which data to *select* and return to the end user.

Whenever you see the word *traversal*, you can associate it with the idea of walking. To us, a graph traversal is a walk through your graph data. When we write graph traversals, we picture walking to and from pieces of graph data in our minds.

Let's go back to the graph query we just wrote to show you how we think of traversals as walking around graph data. In the first part of the graph query, we found a single vertex as our starting place: `customer_0`. From this customer, we needed to walk through the outgoing edge labeled `uses`. We walked through this edge using the `out()` step in Gremlin so that we could arrive at the credit card vertex. Once we were at the credit card vertex, we could look at the properties on the vertex. Specifically, we wanted access to Michael's credit card number: `cc_17`.

> For the best performance, we advise that you always start your traversal from a specific vertex via the full primary key. For Apache Cassandra users, this is the same as providing the full primary key for a CQL query.

It helps to have a copy of Figure 3-13 to look at as you start practicing your first graph traversals. From an image on paper, you can see where you need to start and end. This is just like using a map for navigation, but in this case, you are walking around your data. With graph data, you can use your graph model to find your starting place, find your ending place, and translate the walk between them into your Gremlin statements. With enough practice, you will eventually be able to do all of this in your head.

### Query: Which accounts does this customer own?

The next C360 query for our application wants to know which accounts a specific customer owns. Following the same pattern as before, we are going to access the vertex for `customer_0` and walk to the account vertex. From the account vertex, we can access the unique ID for the account:

```
dev.V().has("Customer", "customer_id", "customer_0").// WHERE
        out("owns").                                 // JOIN
        values("acct_id")                            // SELECT
```

As before, this query follows the `WHERE-JOIN-SELECT` pattern. The first part of this Gremlin query is similar to a *where* statement: `dev.V().has("Customer", "customer_id", "customer_0")`. We say that this step is finding *where* you are starting your graph traversal.

The second step in this traversal is like a *join* statement: `out("owns")`. This step is walking through the `owns` relationship coming out of the customer to *join* the customer to their data. The last step *selects* the data to return to the end user, specifically the account id: `values("acct_id")`. This query will return the following data:

```
"acct_14"
```

Let's try the same query again, but this time we would like to display the customer's name alongside their account ID. To do this, we need to remember the data we have visited as we walked through the graph. This introduces two new Gremlin steps: `as()` and `select()`. The `as()` step is similar to labeling the data as you walk through your graph, like leaving breadcrumbs behind as you walk around a maze.

Once we are done, we can recall the visited data with the other new step: `select()`. We use the `select()` step to return the data from the query:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
          as("customer").                             // LABEL
        out("owns").                                  // JOIN
          as("account").                              // LABEL
        select("customer", "account").                // SELECT
          by(values("name")).                   // SELECT BY (for the customer)
          by(values("acct_id"))                 // SELECT BY (for the account)
```

As before, this query follows the same `WHERE-JOIN-SELECT` pattern, with two additions. This query adds in the need to `SAVE` and `SELECT` specific data points from the query.

Let's walk through the steps in this query.

Once again, we start with *where* we need to go in our graph data, `dev.V().has("Customer", "customer_id", "customer_0")`. We want to remember this data for later, so we save the data with the step `as("customer")`. We continue to follow the pattern as before, *joining* the customer to their account data by walking through the `owns` edge. Now we have arrived at the account vertex. We want to save this vertex by using `as()`, like before. Last, we need to *select* multiple pieces of data. We do this with `select("customer", "account")`.

The remaining two steps that use `by` are important to call out. This step helps us shape the results of our query. After the `select("customer", "account")` step, we have two vertex objects: the customer and account vertices, respectively. Our original query wanted to access the customer's name and account ID. That is where the `by` step comes in. We want to view the customer according to their name and the account according to its ID. The `by` steps are applied in order to the vertex objects.

This query returns the following JSON:

```
{
  "customer": "Michael",
  "account": "acct_14"
}
```

#### Query: Which loans does this customer owe?

So far, we have seen three graph traversals and two different ways to select data from your graph. Next, let's explore the third query for our C360 application. This query wants to access the loans associated to a customer. Let's use `customer_4` for this example since she has multiple loans in our dataset. In this query we just want to look at the loan IDs:

```
dev.V().has("Customer", "customer_id", "customer_4"). // WHERE
        out("owes").                                   // JOIN
        values("loan_id")                              // SELECT
```

This query follows the same WHERE-JOIN-SELECT pattern that we saw in the previous section. This query will return the following data:

```
"loan_18",
"loan_80"
```

### Query: What do we know about this customer?

The final query of a C360 application is to access an individual customer and all of their relevant data. The query for this will start at `customer_0` and walk through *all* outgoing edges that are connected to `customer_0`. Then we return the data from all vertices that are in this first neighborhood of `customer_0`. This query gives us all of the data about `customer_0`:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
        out().                                         // JOIN
        elementMap()                                   // SELECT *
```

This query will return the data shown in Example 3-1.

*Example 3-1.*

```
{
  "id": "dseg:/CreditCard/cc_17",
  "label": "CreditCard",
  "cc_num": "cc_17"
},
{
  "id": "dseg:/Loan/loan_32",
  "label": "Loan",
  "loan_id": "loan_32"
},
{
  "id": "dseg:/Account/acct_14",
  "label": "Account",
  "acct_id": "acct_14"
}
```

Example 3-1 shows everything stored in DataStax Graph about each vertex: an internal `id`, the vertex's `label`, and then all properties. Let's inspect the JSON that describes Michael's credit card. First, there is an `"id": "dseg:/CreditCard/cc_17"`. This is the internal identifier used in DataStax Graph to describe that piece of data. The internal `id` in DataStax Graph is a URI, or Uniform Resource Identifier. Next, we see the vertex's label, `"label": "CreditCard"`. Last, we see the only property we

stored in the graph about credit cards: `"cc_num": "cc_17"`. We interpret the JSON about the loan and account vertices similarly.

These traversals are the base of what is required to extract the data in your C360 application. We recommend keeping a copy of your graph data model nearby when you are first starting to write graph traversals. Once you understand the basic steps, you can use an image of your data model to walk from your starting point to your destination. After some practice, this is an art that you may be able to visualize in your head as if you were the one walking around the data.

We constructed this example to show that graph applications can make data retrieval easier. As seen in this section, there were significantly fewer steps to the query, and they were easier to follow. The adjustment from relational to graph query languages requires an adjustment in your mentality to traversing or walking through your data. The learning curve is steep; we don't want to hide that. However, once you can picture yourself walking through your graph data, writing graph queries can be as simple as learning a new set of tools.

# Relational Versus Graph: How to Choose?

Through the lens of a C360 application, let's consider the benefits and drawbacks of a relational database implementation versus those of a graph database implementation. In considering these two technologies, we are going to compare them in four areas. We are going to examine each technique's approach to data modeling, representing relationships, and query languages.

## Relational Versus Graph: Data Modeling

There are quantitative and subjective items to consider when comparing the differences in data modeling for relational or graph databases. The quantitative arguments around data model design will point toward a relational system as the clear winner due to the higher volume of resources and production usage of relational systems. The techniques, tricks, and optimizations for a relational system are very well documented and accessible for all members and abilities within a development team.

On a more subjective note, data modeling techniques with graph technologies are significantly more intuitive. Specifically, when using graph technology, the human-to-computer translation of data is preserved; the way you think about your data is nearly the same as the way you would represent it digitally in a computer. This shorter translation from human intuition to machine representation allows you to extract deeper insights about the relationships in your data. This arguably makes graph technology easier to use over the system design required for the same implementation in a relational database.

## Relational Versus Graph: Representing Relationships

There has been and continues to be a growing demand for modeling and storing relationships within a database. This has created both good and bad news for relational systems. The good news, as stated before, is that the tips, tricks, and techniques for modeling relationships with relational technology are well documented. Adding relationships to an existing relational database can be as simple as adding a join table or foreign key constraint. With the new join table or foreign key, relationships are queryable and accessible. Essentially, getting the data into the system is well documented and relatively easy for the developer.

On the downside, getting the relationships back out of a relational system in a meaningful way has a much steeper curve; it is very difficult to reason about the relationships stored in a relational system because of the large gap from idea to implementation to machine. The process from conversation to modeling to reasoning is much more disconnected with relational technology than it is with graph technology. The disconnect lies in the mental transformation required to map your human understanding of data into relational models and down to tables. This translation requires significant mental interpretation to follow and reason about relationships within the data stored in a relational database.

Graph technologies were created from this gap. If there is a need to model and reason about relationships in your data, graph technologies provide a more seamless transition from human understanding to machine representation of your data and back. The crux of this stance is whether or not relationships exist within your data and are useful for deeper analytics and reasoning. If you need to model and reason about relationships in your data, then graph technologies are the way to go.

## Relational Versus Graph: Query Languages

There are three aspects we would like to examine when comparing query languages for the two systems: language complexity, query performance, and expressiveness.

First, let's talk about what we mean by language complexity. After designing relationships into your system, the query language introduces additional complexity to your evaluation of the database within your architecture. At this level, it is the query language that will highlight all of the complexities or simplifications that were made during the implementation process. The additional complexity is experienced as queries are developed and lengthened to pull together the required data.

Teams often measure query language complexity by query development time, maintainability, and ease of transferring knowledge. When you are considering SQL and Gremlin, these comparisons come down to adoption maturity and personal preference. SQL is the clear winner in language maturity. However, we see the scales tip toward Gremlin for deeply nested queries or those requiring a large number of joins.

The next evaluation of query languages measures query performance. Query performance measures a multifaceted and complex dependency of database-tuning exercises from indexing, partitioning, load balancing, and more optimizations than will fit in this book.

When we are considering the scope of a C360 application in a small deployment, it is likely that the queries against a properly indexed relational system will consistently outperform the same queries in a graph database. This is because the queries for the simplistic C360 application are very shallow graph queries; the queries stay within the first neighborhood of the customer. As graph queries get deeper, like what we will see in the next chapter, the performance debate between graph technology and relational technology heavily favors graph solutions.

The last comparison to make considers query language expressiveness. In our experience, the expressiveness of graph query languages solidifies the power of using graph data in an application. The difference in query complexity between the two systems illustrates that a more expressive language like Gremlin is a significant improvement for querying relationships in your system. Graph query languages on top of graph databases allow for a significant reduction in the code required to access and extract relationships from data. Only time will allow graph technologies to mature to the same levels as relational standards.

## Relational Versus Graph: Main Points

For a loose summation, the points that we can make for each option, see Table 3-2.

*Table 3-2. A summary of considerations when choosing a relational or graph database for a C360 application*

| | Relational | Graph Databases |
|---|---|---|
| Data modeling | Well documented | Digital representation matches human interpretation |
| Representing relationships in data | Known limitations and complexities | More intuitive representation |
| Queries | Well documented | Steep learning curve |
| | Difficulty when querying many relationships together | More expressive query language |

For any area in which you can compare these two technologies, the advantages and disadvantages for either choice come down to maturity. The adoption, documentation, and community are much more evolved for relational technologies than for graph technologies. This maturity likely translates to lower risk and faster execution for traditional applications. Today, graph technologies cannot compete with relational in the categories of maturity and time to delivery for a new application.

On the other hand, relational technology is reaching its limits for delivering valuable insights into relationships within data. This is a significant problem because relationships naturally occur within data and are instrumental in delivering improved insights into your business. In this regard, graph technology is the better option for applications that require relationships to make business decisions. It is the best choice for delivering and reasoning on relationships within your data, which is not achievable at depth and scale with relational databases.

# Summary

The power and vision of implementing a C360 application with graph technology is directly correlated to your business's need for accessing related data across your organization.

Let's unpack what we mean by that.

We have consulted with many enterprises that made specific technology choices over the past decade that in turn led to the construction of data silos. These data silos separated the data relevant to the core entities of their business, such as the customer. From there, recent approaches led to the integration of important data into large monolith systems, such as data lakes. The pain points here were not in the integration of the company's data but in its accessibility.

Who wants to spend time and resources fishing for valuable data in a data lake instead of using a system designed to retrieve valuable data?

For these enterprises, the advent of graph thinking has guided the next iteration of their data architecture. Their goal is to build with technologies that make their data available and representative of their customers' experience. This combination of availability and representation has been and continues to be the driving momentum behind graph technology.

Graph technologies are enabling the next iteration of enterprise data architectures in a way that was previously unachievable. We delved into one version of graph data management in this chapter. Namely, we explored the application and implementation details of a Customer 360 application, a customer-specific use case for graph technology. However, this same template for building data-centric applications with graph technology applies to non-customer-facing applications.

We have seen companies build similar systems around the businesses they interact with—kind of like a Business 360. The applications that organize and deliver all information about important interactions within their business are saving significant overhead in cross-departmental communication. For example, imagine all of the different departments you have to collaborate with in your company to find out the most recent interaction between your company and another vendor. The information for

this request is spread across finance, marketing, sales, customer relations, and likely other departments. The solution to this B2B problem requires the same template as we have described throughout this chapter.

Given the vision for this style of application, the next criteria to evaluate are the time and cost of implementation. These choices likely involve comparing vendors and existing tools, such as using relational or graph technology for your C360 application.

## Why Not Relational?

We get this question all the time: "I can build a C360 application with an RDBMS, so why not use what I already know?"

The short version of our response to this question: relational is great for tabular data, graph is better for complex data. Otherwise, the two are remarkably similar. At its root, your choice comes down to the complexity of your data and what value you want to get out of it.

In the longer version, the key is how your business values time: time spent on engineering custom solutions and time spent on waiting for queries. The differences are quite clear when your business needs to answer deeper or unplanned queries. Relational systems require architectural changes, adding tables, and building your own query languages. Graph systems require augmenting your schema and inserting more data.

Essentially, graph technology makes it easier to work with complex data, whereas relational technology is easier for simple (i.e., tabular) data. The depth and complexity your project needs to expand into will help make this choice clearer for you.

## Making a Technology Choice for Your C360 Application

The decision between relational and graph technology ultimately comes down to your C360 application's full scope. Generally speaking, our experiences have shown us that if your application aims only to unify disparate data sources, then you will achieve the best results from properly tuning a relational system. This realization and commitment to the sole function of your application will save development resources and more quickly get to final delivery for the production system.

On the other hand, if a data management solution or C360 application is a starting point for your data architecture, then the steep learning curve to graph databases will deliver more value in the long run. Graph technologies enable more intuitive reasoning about the relationships that exist across your data. The business objectives that require insight into relationships also require graph technology behind them.

Let us be clear on our points here. The example in this chapter is incredibly primitive. Anything more realistic and more elaborate starts to become a stretch for RDBMS.

And realistic data contains elaborate relationships within it. If your business needs access to these relationships, then you need graph technology.

> If you are going to just build a simple C360 system and nothing more, use relational technology. If you want to understand and explore the connectedness within your data, use graph technology. There are pluses and minuses for each choice, but for the scenario we have set up in this chapter, graph technology is the winner.

Whichever data problem your business faces, be aware that those teams with the need to build *and extend* a foundation are turning to graph technology. A successful integration of graph technology into your architecture needs to start with a C360 application as its foundation and build from there. With a C360 application as a foundation, your business is set up to go after deeper graph traversals for more valuable insights from your data. In the next chapter, we are going to extend our simplistic C360 application to a more complete scenario that will highlight how graph technology and RDBMS diverge in terms of ease of use and time to market.

CHAPTER 4

# Exploring Neighborhoods in Development

To get to the next phase in graph application development, we are going to build upon the simple Customer 360 (C360) application from Chapter 3. We'll add a few more layers, or neighborhoods, onto that example to illustrate the next wave of concepts in graph thinking.

Adding data to our example provides a more realistic picture of the complexity of data modeling, querying, and applying graph thinking to our customer-centric financial data.

We consider the transition from the basic example in Chapter 3 to the complexity in this chapter to be analogous to steps in the process of learning how to scuba dive. What we did in Chapter 3 was like starting to learn how to scuba dive in a wading pool; it is not really clear what the point is when you are in water that shallow. But we needed to start from a familiar place. The examples in this chapter are like scuba diving in a deep pool. Afterwards, we will be ready to head into more interesting depths in Chapter 5.

## Chapter Preview: Building a More Realistic Customer 360

There are three main sections within this chapter.

In the first section, we will explore and explain graph thinking to present best practices in graph data modeling. We will do this by adding more neighborhoods of data to our C360 example so that we can answer the following questions:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?

3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan. (Query 3 is an example of personalization.)

Throughout this initial section, we will follow query-driven design to illustrate common best practices for creating a property graph data model. Topics include mapping your data to vertices or edges, modeling time, and common mistakes.

In the next section, we will build up deeper Gremlin queries. These queries walk through three, four, and five neighborhoods of data. We will also introduce how to use properties to slice, order, and range over graph data, and we will discuss querying in time windows. By the end of this section, we will have illustrated all of the data, technical concepts, and data modeling that we planned for our example.

We will end the chapter by revisiting the basic queries to introduce some more advanced querying techniques. These techniques are most commonly a part of trying to format your query results into a more user-friendly structure.

This content sets us up to present the final, production-quality schema for this example, which we will do in Chapter 5.

## Graph Data Modeling 101

During the early days of working with graph databases backed by Apache Cassandra, my team was sitting around the couches in the living room of our venture-backed startup. We were whiteboarding a graph data model for storing healthcare data in a graph database.

We quickly agreed that doctors, patients, and hospitals were our primary entities of importance, and therefore they would be vertices. Everything else after that was a debate. Vertices, edges, properties, and names: everyone had a defensible opinion about everything. Our most memorable disagreements were polarizing. What should we name the edges between doctors and patients? All of these entities live or work somewhere; how do we model addresses? Is country a vertex or a property, or should it be left out of our model?

It was a difficult conversation. It took much longer than we had expected to arrive at a design consensus, and none of us really felt comfortable with it.

Since that design session, each time I advise a graph team around the world, I can feel similar tensions and see similar design consensus. The tensions are always real, always there, and always observable.

This section is all about helping your team have a more constructive discussion about your graph data model. To accomplish this, we want to walk through three sections of advice for creating a good graph data model. Those sections of advice will be:

1. Should This Be a Vertex or an Edge?
2. Lost yet? Walk Me Through Direction.
3. A Graph Has No Name: Common Mistakes in Naming

We selected these topics for two reasons. First, these topics cover most of the points of contention you will encounter during the modeling process. Second, these topics support where we are in the development of the running example for these chapters. Details for deeper and more advanced modeling advice will be introduced when we get there.

## Should This Be a Vertex or an Edge?

This is the most debatable topic about property graph modeling. From the middle of the most heated debates, we've grabbed a number of tips for creating graph data models.

Let's start our tips at the beginning. In our world, the beginning is where you want to start your graph traversals.

**Rule of Thumb #1**

If you want to start your traversal on some piece of data, make that data a vertex.

To unpack our first tip, let's revisit one of the queries we constructed in Chapter 3:

```
Which accounts does this customer own?
```

There are three pieces of data required to answer that question: customers, accounts, and a connection from which customer owns an account. Think about how you could use that data to "find all accounts owned by Michael." There are two ways to translate this statement into a database query: "Michael owns accounts" or "accounts owned by Michael."

Let's talk about the first option: starting with Michael to find his accounts. This means that you are starting with data about people—specifically, the piece of data about Michael. In your head, when you find a starting place for a query, you would want to translate that data to being a vertex label in your graph model. With this, we have our first vertex label for our graph model: customers.

Consider the second way to find this information: you could first find all accounts and then keep only those that are owned by Michael. In this case, you are starting with the data about accounts. Now we have a second vertex label for our graph model: accounts.

This sets us up for the next tip on how to find the edges in your data.

> **Rule of Thumb #2**
>
> If you need the data to connect concepts, make that data an edge.

For the query we are working with, we know that Michael will be a vertex label and that his account is another vertex label. That leaves the concept of ownership, and yes, you guessed it—it will be the edge. The concept of ownership links a customer to an account for our example data.

To find the edges in your model, examine your data. You find your edges from within the information that links concepts together *and* to which you have access.

When working with graph data, these edges are the most important piece of your graph model. Edges are why you need graph technology in the first place.

Putting these two together, you can derive the following rule for labeled property graph models.

> **Rule of Thumb #3**
>
> Vertex-Edge-Vertex should read like a sentence or phrase from your queries.

Our advice here is to write out how you want to query with your data into short phrases like "customer owns account." Identifying these queries and phrases remains a simple way to identify how you want to map your data into graph objects in a property graph database, as shown in Figure 4-1.

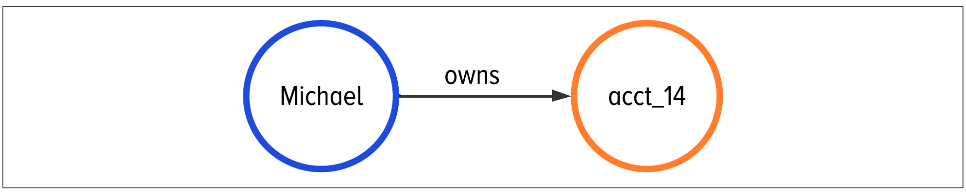



*Figure 4-1. Two vertices, named Michael and* `acct_14`*, with an edge (relationship) titled,* `owns`*; this illustrates an example of translating short phrases of noun-verb-noun to a property graph model: Michael owns account 14*

Generally speaking, written forms of your graph queries will translate verbs to edges and nouns to vertices.

This isn't the first time the graph community has worked with semantic phrases and graph data. Those of you from the semantic community are likely shouting, "We've seen this before!" And you are right; we have.[1]

Putting recommendations #2 and #3 together yields a specific way to translate how you think into graph objects.

#### Rule of Thumb #4

Nouns and concepts should be vertex labels. Verbs should be edge labels.

Depending on how you think, there are times at which tip #3 and tip #4 can create ambiguous scenarios. We want to delve into some semantics here to help you navigate different ways that people see and think about data.

Specifically, if you think "Michael owns an account," then "owns" should be an edge label. This is a case in which you are thinking actively about the relationship between Michael and his account. And this active line of thought translates `owns` to a verb that connects two pieces of data together. This is how we arrive at "owns" as an edge label.

However, there are cases in which you may see this same scenario differently. Namely, if you are thinking "We need to represent the concept of ownership between Michael and his account," then ownership should be a vertex label. In that case, you are thinking of ownership as a noun—that is, an entity. The difference is that in this case, it is likely that the ownership needs to be identifiable. You probably are trying to relate ownership in other ways. In these cases, other questions you may plan on asking are, "Who established that ownership?" or "Who does the ownership transfer to if the primary agent dies?"

We acknowledge that we are getting into the weeds here. But we know that you will eventually find yourself in the weeds as well. We hope that the guidance we are providing will help you find your way back up and out.

Our first four tips introduced the fundamentals for identifying vertices and edges in your graph data. Let's walk through how to reason about the direction of your edge labels.

1 Ora Lassila and Ralph R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," 1999. *https://oreil.ly/zWcnO*

## Lost Yet? Let Us Walk You Through Direction

The questions and queries for this chapter integrate more data into our model. Specifically, we want to add transactions into our data so that we can answer questions like:

```
What are the most recent 20 transactions involving Michael's account?
```

To answer this query, we need to add transactions into our data model. And these transactions need to give us a way to model and reason about how transactions withdraw and deposit money between the accounts, loans, and credit cards.

When you first start writing graph queries and iterating on data models, it is very easy to get turned around in your data model. Direction of an edge label is a difficult thing to reason around, which is why we make the following recommendation.

#### Rule of Thumb #5

When in development, let the direction of your edges reflect how you would think about the data in your domain.

Tip #5 infers the direction of an edge label as you combine and apply the advice from the previous four tips. At this point, the pattern of Vertex-Edge-Vertex should be easily read as subject-verb-object sentences.

Therefore, the edge label's direction comes from subject and goes to object.

Coming up with edge labels between transactions is a discussion we have seen play out many times. Let's follow through our thought process to detail how we reasoned about modeling something like a transaction in a graph.

### An evolution of modeling transactions in a graph

Think about how you would first add transactions into your graph model. You likely are thinking about how an account transacts with other accounts, or something like we are showing in Figure 4-2.

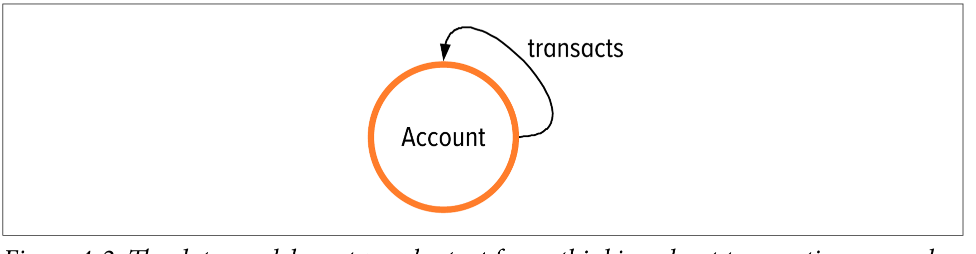



*Figure 4-2. The data model most people start from: thinking about transactions as verbs, with phrases like "this account transacts with that account"*

The model for Figure 4-2 doesn't work for our example because it uses the idea of a transaction as a verb, whereas our questions use transactions as nouns. We want to know things like an account's most recent transactions and which transactions are loan payments. In this light, we are really thinking about transactions as nouns.

Therefore, transactions need to be vertex labels in our example.

Now we need to reason about the direction of the edges. Most people start with modeling edge direction to follow the flow of money, as shown in Figure 4-3.

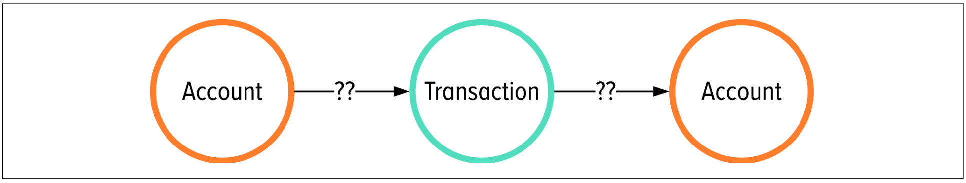



*Figure 4-3. Modeling edge direction according to the flow of money*

The challenge with a model like Figure 4-3 is to come up with intuitive names for the edges that make it easy to answer our chapter's questions. The edge direction in Figure 4-3 models the flow of money and is awkward for how we are using transactions in our questions. Would we say, "This account had money withdrawn from it via this transaction"? Let's hope not.

So Figure 4-3 isn't going to work for our example, either.

Let's recall our chapter's questions and reason about how we use transactions in the queries. We came up with the following subject-verb-object sentences for the context in which we are using transactions in our example:

1. Transactions withdraw from accounts.
2. Transactions deposit to accounts.

These two phrases might work; let's see how this would work with data. In our data, we could model a transaction and how it interacted with accounts as shown in Figure 4-4.

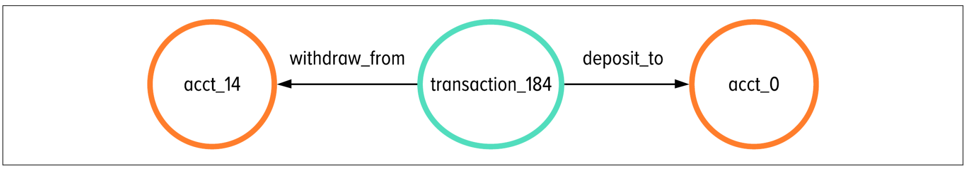



*Figure 4-4. Modeling the direction of your edges according to how you would use them in your queries*

For the example in this chapter, we think that Figure 4-4 makes it reasonably easy to use our model to answer our questions. This gives us direction for both of our labels: the edge labels will flow from a `Transaction` and go to an `Account`. The schema is shown in Figure 4-5.

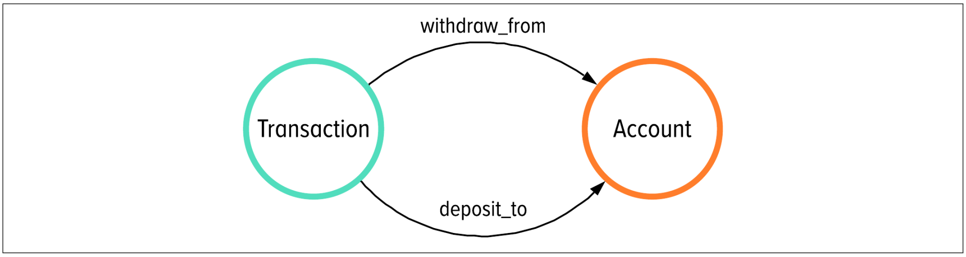



*Figure 4-5. Modeling the direction of your edges according to how you would think about the data in your domain*

By breaking down your queries into short, active phrases of the structure subject-verb-object, you will be able to naturally find what needs to be a vertex or edge label in your graph model. Then the edge label's direction will come from the subject and go to the object.

Let's zoom out from the nuances of modeling direction for transactions and get back to the final main element of a graph's schema: properties.

#### When do we use properties?

Let's repeat the first query that will use the transaction vertices:

```
What are the most recent 20 transactions involving Michael's account?
```

The short version of our query from above translates to the following short phrases:

1. Michael owns account
2. Transactions withdraw from his account
3. Select the most recent 20 transactions

So far, we can walk through customers, accounts, and transactions within our graph. Now our question asks for the 20 most recent transactions from an account. This means that we need to subselect our transactions to include only the most recent ones.

Therefore, we will want the ability to filter transactions by time. This brings us to our last tip related to data modeling decisions.

**Rule of Thumb #6**

If you need to use data to subselect a group, make it a property.

Ordering transactions by time requires us to have that value stored in our graph model: enter properties. This is a great use of a property on the transaction vertex so that we can subselect those vertices in our model. Figure 4-6 shows how we would add time into our ongoing example.

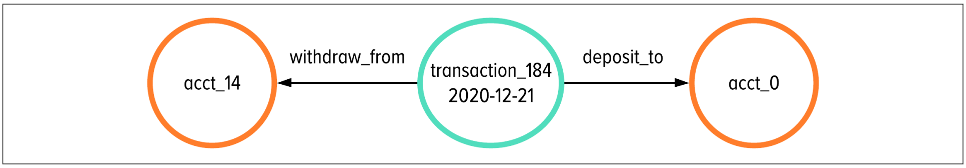



*Figure 4-6. Modeling time as a property on the transaction vertex so that we can subselect to query for only the most recent transactions*

Together, tips #1–6 give you a great starting point for identifying what will be a vertex, an edge, or a property in your graph data model. We have one last section of data modeling best practices to consider before we start the implementation details for this chapter.

## A Graph Has No Name: Common Mistakes in Naming

The callouts in the upcoming section are common mistakes. Each mistake is followed by our bad-better-best recommendations.

Arriving at a consensus on what something should be named and maintained with your codebase is surprisingly difficult. We have three topics on which teams commonly waste their valuable time in bikeshedding how to address naming conventions in their graph data model.

**Pitfalls in Naming Conventions #1**

Using the word `has` as an edge label.

One of the most common mistakes we see comes from naming all of your edges with the label `has`, as shown on the left side of Figure 4-7. This is a mistake in naming because the word `has` does not provide meaningful context regarding the edge's purpose or direction.

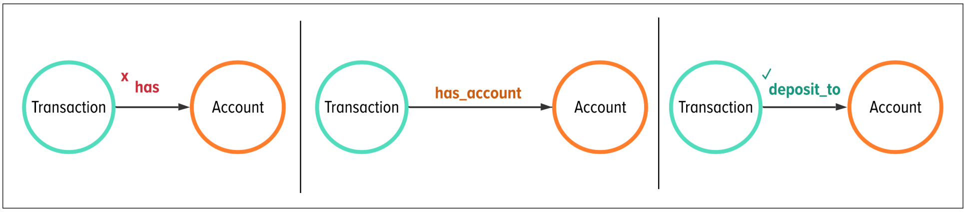



*Figure 4-7. From left to right: the bad, better, and recommended ways to name your edges*

If your graph model uses `has` for its edge labels, we have two recommendations for you. A better edge label would have the form `has_{vertex_label}`, as shown in the center in orange in Figure 4-7. This type of name allows you to have more specificity in your graph queries while also providing a more meaningful name to maintain in your codebase.

The preferred solution to this problem is shown in green at far right in Figure 4-7. This recommendation advises you to use an active verb that communicates meaning, direction, and specificity to your data. We are going to use the edge labels `deposit_to` and `withdraw_from` to connect transactions to the accounts in our examples.

After meaningful edge labels have been selected, it is also a common mistake to create property names that do not help uniquely identify your data. This brings us to our next pitfall in property graph modeling.



**Pitfalls in Naming Conventions #2**

Using the word `id` as a property.

The concept of which pieces of data uniquely identify an entity is a deep topic. Using a property key called `id` is a bad decision because it is not descriptive of what it is referring to. Additionally, `id` is a naming clash with the internal naming conventions within Apache Cassandra and is not supported in DataStax Graph.

A slightly better convention would be to name the property that uniquely identifies your data with `{vertex_label}_id`, as shown at center in Figure 4-8. We use this a few times throughout the book because we are working with synthetic examples, and this type of identifier is perfectly fine if you use randomly generated identifiers, like UUIDs (universally unique identifiers). However, you will see us move to using more descriptive identifiers when we work with open source data. These identifiers represent concepts that uniquely identify entities within their domain, such as social security numbers, public keys, and domain-specific universally unique identifiers.

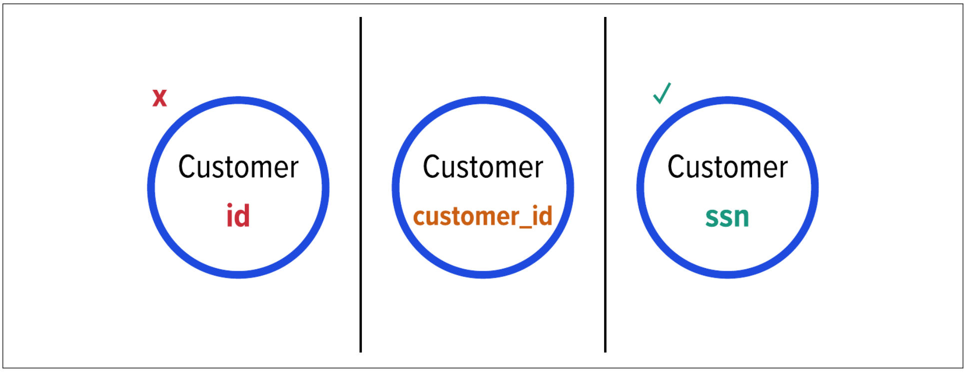



*Figure 4-8. From left to right, the bad, better, and recommended ways to name a property to uniquely identify your data.*

This brings us to the last and debatably most important mistake that we see throughout application codebases.

**Pitfalls in Naming Conventions #3**

Inconsistent use of casing.

When it comes to casing, the best approach follows the language conventions that you are writing in. Some languages have style guides that promote `CamelCase`, whereas others prefer `snake_case`. For the examples in this book, we plan to follow the following casing and styles:

1. Capital `CamelCase` for vertex labels
2. Lowercase `snake_case` for edge labels, property keys, and example data

This last tip feels a bit pedantic to even bring up in a graph book. We are mentioning it because consistency in naming conventions tends to be forgotten, creating expensive roadblocks for teams during the last stretch of getting their graph technology into production. The more trivial these tips seem to your team, the better off you probably already are in making sure to remember them.

## Our Full Development Graph Model

The previous discussion of graph data modeling illustrated how we broke down our first query to evolve the example from Chapter 3. In this section, we want to build up the remaining elements in our data model to answer all the questions for this chapter's example.

The example in this chapter adds schema and data that enable our application to answer the following three questions:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?
3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

We have already stepped through how to model the first question. Let's take a closer look at it.

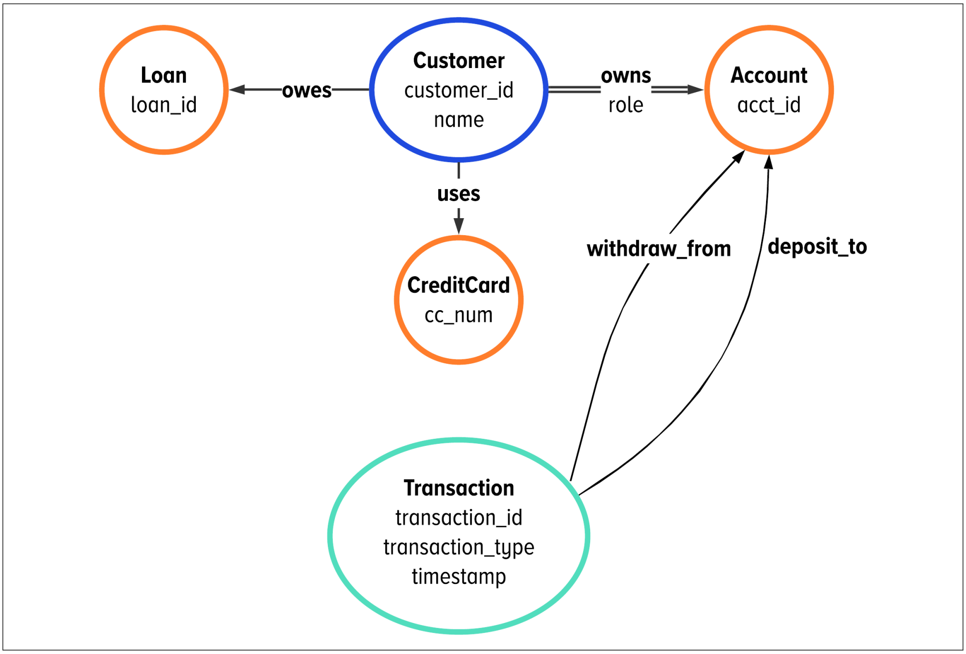



*Figure 4-9. The augmented graph schema from Chapter 3 that applies the data modeling principles to answer the first query of our expanded example*

The graph schema in Figure 4-9 applies the principles we built up to answer the first question into a graph data model. The new vertex label is `Transaction`, with two new edge labels to the `Account` vertex: `withdraw_from` and `deposit_to`, respectively. We discussed how and where to model time in our graph, which you see in Figure 4-9 with `timestamp` on the `Transaction` vertex.

Next, let's consider this chapter's remaining questions for our example in this chapter by modeling the queries:

```
1. In December, at which vendors did Michael shop, and with what frequency?

2. Find and update the transactions that Jamie and Aaliyah most value:
     their payments from their account to their mortgage loan.
```

To arrive at a data model for these questions, let's apply the thought processes we introduced in "Graph Data Modeling 101" on page 82. Following the advice there, we came up with three statements about transactions:

1. Transactions charge credit cards.
2. Transactions pay vendors.
3. Transactions pay loans.

From these statements, we can find the rest of our required schema elements. First, we need a new vertex label to represent where our customers shop: `Vendor`. Next, we need an edge label, `pay`, for a transaction to the `Loan` or `Vendor` vertex labels. Last, we need another edge label, `charge`, to indicate that a transaction charges a credit card.

Bringing all of this together, we have the schema shown in Figure 4-10.

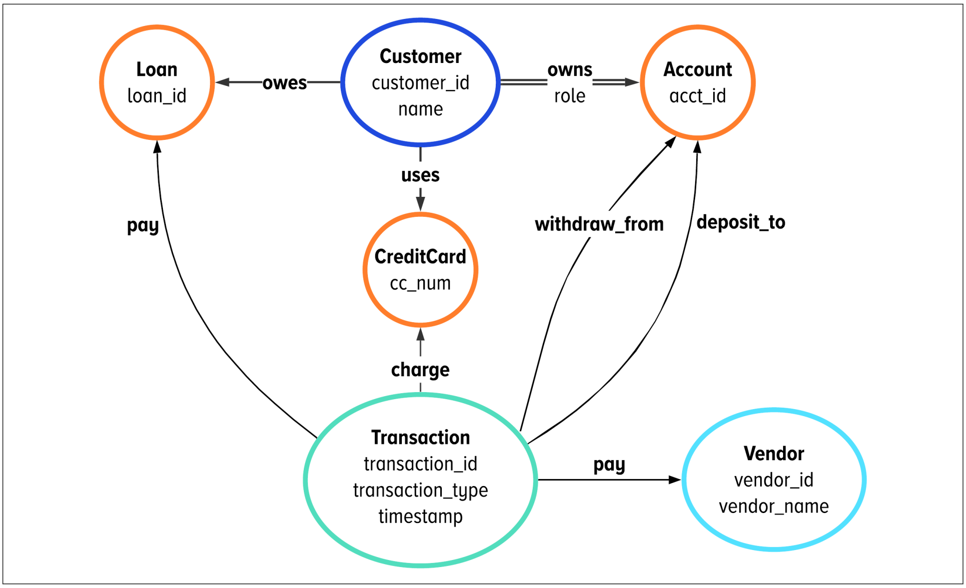



*Figure 4-10. The development schema that answers all of the queries we aim to build for this example*

## Before We Start Building

We reduced the full perspective on graph data modeling to include only the practices that we need for our current example. Beyond these core principles, you will find

edge cases about your data that are not covered here. That is expected. We are teaching a thought process and selected the principles here as a starting guide for modeling your data like a graph.

> If we could ensure you understood one concept about graph data modeling, it would be the following: modeling your data as a graph is just as much of an art as it is engineering. The art of the data modeling process involves creating and evolving your perspective on your data. This evolution translates your mindset into the paradigm of relationship-first data modeling.

When you find new modeling cases in this book or in your own work, ask the following questions about what you are modeling to help develop your own reasoning:

1. What does this concept mean to the end user of the application?
2. How are you going to read this data in your application?

Defining your data model is the first step in applying graph thinking to your application. Focus on the data you can integrate, the queries you want to ask, and what this will mean to your end user. When combined, those three concepts articulate how we see, model, and use graph data within an application.

## Our Thoughts on the Importance of Data, Queries, and the End User

To help you learn and apply our perspective to building your own graph model, let's walk through the importance of data, queries, and the end user.

Our first piece of advice is to focus on the data you have. It is easy to boil the ocean by modeling your industry's entire graph problem; avoid this rabbit hole! Your graph model will evolve if you keep centered on getting to production with the data with which your application will be working.

Second, apply the practice of query-driven design. Build your data model to accommodate only a predefined set of graph queries. A common red herring we run into on this topic is those applications that aim to create open traversals across any discoverable data in a graph. For developmental purposes, the ability to explore and discover makes sense. However, for production use, an application with open traversal access can introduce a myriad of concerns.

For security, performance, and maintenance implications, we strongly advise teams not to create production platforms with unbounded and unlimited traversals. The warning sign we see is a lack of specificity for your graph application. We know this perspective is very hard to apply when you are first exploring graph data. We see the

line here as setting expectations between what you want to do during development versus what you want to push to production in a distributed production application.

Last and most importantly, you have to consider what the data means to your end user. Everything from selecting naming conventions to the objects in your graph will be interpreted by someone else: your team members or your application users. Naming conventions and graph objects are interpreted and maintained by your engineering team members; choose them wisely.

Ultimately, your graph data will be presented to an end user through your application. Spend time designing your data architecture, models, and queries to present information that is most meaningful to them.

When combined, these three concepts articulate how we see, model, and use graph data within an application. Again, the three concepts are to build with the data you have, follow query-driven design, and design for your end user. Following these design principles will help get you unstuck during those difficult data modeling discussions and prepare your application to be the best use of graph data the industry has ever seen.

## Implementation Details for Exploring Neighborhoods in Development

Our schema from Figure 4-10 requires only two new vertex labels: `Transaction` and `Vendor`. What you have practiced a few times prior to now is how to take a schema drawing and translate it into code. We showed the schema in Figure 4-10, and in Example 4-1 we show you the code.

*Example 4-1.*

```
schema.vertexLabel("Transaction").
       ifNotExists().
       partitionBy("transaction_id", Int).
       property("transaction_type", Text).
       property("timestamp", Text).
       create();

schema.vertexLabel("Vendor").
       ifNotExists().
       partitionBy("vendor_id", Int).
       property("vendor_name", Text).
       create();
```

In case you are wondering, we are using `Text` as the data type for timestamp to make it easier to teach concepts in our upcoming examples. We will be using the ISO 8601 standard format stored as text.

In addition to these vertex labels, we added relationships between the `Transaction` vertex and the other vertex labels in this graph. Let's start with the new edge labels between the `Transaction` and `Account` vertex labels. The schema code for the new edge labels is shown in Example 4-2.

*Example 4-2.*

```
schema.edgeLabel("withdraw_from").
       ifNotExists().
       from("Transaction").
       to("Account").
       create();

schema.edgeLabel("deposit_to").
       ifNotExists().
       from("Transaction").
       to("Account").
       create();
```

These two edges model how money moves to and from an account within your bank. In Example 4-3, we add in the rest of the edge labels in our example:

*Example 4-3.*

```
schema.edgeLabel("pay").
       ifNotExists().
       from("Transaction").
       to("Loan").
       create();

schema.edgeLabel("charge").
       ifNotExists().
       from("Transaction").
       to("CreditCard").
       create();

schema.edgeLabel("pay").
       ifNotExists().
       from("Transaction").
       to("Vendor").
       create();
```

These last three edge labels complete the edges we will need to describe transactions between the assets in our example.

## Generating More Data for Our Expanded Example

As examples grow, so too does the data. We wrote a small data generator to expand the data from Chapter 3 to include our data model from Figure 4-10. If you are interested in the data generation process for this chapter, you have two options.

Your first option is to use the bash scripts to reload the exact same data you will see in the upcoming examples. We will teach you about this tool and process in Chapter 5, but you are welcome to preview the loading script in the GitHub repository. We recommend using the scripts throughout this book if you would like the examples you are running locally to match the results we show in the text.

Your second option is to dive into and execute our data generation code. We provided our code in a separate Studio Notebook called `Ch4_DataGeneration`. We recommend this option if you want to dig into creating fake data with Gremlin and the methods we used.

> **An Important Warning About the Data Generation Process**
>
> If you rerun the data insertion process in your Studio Notebook, the results in your local graph will not precisely match the results printed in this text. If you want the data to match precisely, we recommend importing the exact same graph structure via DataStax Bulk Loader. You will find all of this in the accompanying technical materials.

Up to this point, we have accomplished many tasks. We explored our first set of data modeling tips, created a development model, looked at the schema code, and inserted data.

The last main task is to use the Gremlin query language to walk around our model and answer questions about our data.

# Basic Gremlin Navigation

The main objective of this chapter is to illustrate a real-world graph schema that walks through multiple neighborhoods of graph data.

> For your reference, we will use the words *walk*, *navigate*, and *traverse* interchangeably throughout this book to mean that we are writing graph queries.

Everything in this chapter up until now was required to set up answering the following three questions in this section:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?
3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

Let's walk through the queries and their results. Then, in the chapter's final section on Advanced Gremlin, we will delve a bit deeper into how to shape the result payload.

Our recommendation is that you find a way to reference Figure 4-10 as you practice the queries in the upcoming sections. We recommend doing this because your schema functions as your map; you need to know where you are so that you can walk in the right direction to your destination.

### Query 1: What are the most recent 20 transactions involving Michael's account?

Let's start with some pseudocode in Example 4-4 to think about how we are going to walk through our data to answer this first question.

*Example 4-4.*

```
Question: What are the most recent 20 transactions involving Michael's account?
Process:
    Start at Michael's customer vertex
    Walk to his account
    Walk to all transactions
    Sort them by time, descending
    Return the top 20 transaction ids
```

We used the process outlined in Example 4-4 to create the Gremlin query in Example 4-5.

*Example 4-5.*

```
dev.V().has("Customer", "customer_id", "customer_0"). // the customer
        out("owns").                         // walk to his account
        in("withdraw_from", "deposit_to"). // walk to all transactions
        order().                             // sort the vertices
          by("timestamp", desc).   // by their timestamp, descending
        limit(20).                           // filter to only the 20 most recent
        values("transaction_id")             // return the transaction_ids
```

A sample of the results:

```
"184", "244", "268", ...
```

Let's dig into this query one step at a time.

On line 1, `dev.V().has("Customer", "customer_id", "customer_0")` looks up a vertex according to its unique identifier. Then on line 2, the step `out("owns")` walks through the outgoing `owns` edge to the `Account` vertices for this customer. In this case, Michael has only one account.

At this point, we want to access all transactions. On line 3, the `in("withdraw_from", "deposit_to")` step does just that: we walk through the incoming edge labels to access transactions. At line 4, we are on the transaction vertices.

> We left a detail out of "An evolution of modeling transactions in a graph" on page 86 that we want to bring up now. The simplicity of line 3 in Example 4-5 was also part of the motivation that led to how we designed the edges in our data model. This first query was much harder to write and reason about when the edges were going in different directions.

The `order()` step on line 4 indicates that we need to provide some sort of order to the vertices, which are transactions. We specify the sort order on line 5 with the `by("timestamp", desc)` step. This means that we are going to access, merge, and sort all `Transaction` vertices according to their timestamp. Then we want to select only the 20 most recent vertices with `limit(20)`. Last, on line 7, we want to get access to the `transaction_ids`, so we select them via the `values("transaction_id")` step.

This query will return a list of values that contains the `transaction_id` for each of the 20 most recent transactions across all of the customer's accounts.

Imagine how much more powerful this would be to display for the end user. They would be able to see the details that are most relevant to them instead of navigating multiple screens to join this data together in their head. This type of query is vital in understanding how to personalize your application to what a customer most cares about.

### Query 2: In December 2020, at which vendors did Michael shop, and with what frequency?

For this second question, let's start with an outline of the query in Example 4-6 to think about how we are going to walk through our data to answer the question.

*Example 4-6.*

```
Question: In December 2020, at which vendors did Michael shop, and with what frequency?
Process:
    Start at Michael's customer vertex
    Walk to his credit card
    Walk to all transactions
    Only consider transactions in December 2020
    Walk to the vendors for those transactions
    Group and count them by their name
```

We start the process outlined in Example 4-6 in Example 4-7 and complete it in Example 4-8. In preparation for this query, we used the ISO 8601 timestamp standardization in our data to make it easier to range on dates. In the ISO 8601 standard, timestamps are commonly formatted as `YYYY-MM-DD'T'hh:mm:ss'Z'`, where `2020-12-01T00:00:00Z` represents the very beginning of December in 2020.

*Example 4-7.*

```
1 dev.V().has("Customer", "customer_id", "customer_0"). // the customer
2         out("uses").                         // Walk to his credit card
3         in("charge").                        // Walk to all transactions
4         has("timestamp",                     // Only consider transactions
5             between("2020-12-01T00:00:00Z",  // in December 2020
6                     "2021-01-01T00:00:00Z")).
7         out("pay").                          // Walk to the vendors
8         groupCount().                        // group and count them
9           by("vendor_name")                  // by their name
```

The results are:

```
{
  "Nike": "2",
  "Amazon": "1",
  "Target": "3"
}
```

> Randomization affects the results of query 2. If you use the data generation process instead of loading the data, your graph may have a slightly different structure and therefore different counts for query 2.

The setup for Example 4-7 follows a similar access pattern as before, where we start at a customer and then traverse to a neighboring vertex. We start at `customer_0` and walk to their credit cards and then to transactions. On lines 4 through 6, we are using a way to filter your data during a traversal. Here, we are filtering all vertices according

to their timestamps in a specific range. Specifically, `has("timestamp", between("2020-12-01T00:00:00Z", "2021-01-01T00:00:00Z"))` sorts and returns all transactions that have a timestamp during the month of December in the year 2020.

At line 7, following our schema, we walk to the vendors with the `out("pay")` step. Finally, we want to return the vendor's name along with how many times a transaction was observed with that vendor. We do this on lines 8 and 9 with `groupCount().by("vendor_name")`.

In addition to `between`, Table 4-1 lists the most popular predicates you can use to range on values. Please refer to the book by Kelvin Lawrence for the full table of predicates.[2]

*Table 4-1. Some of the most popular predicates that you can use to range on values*

| Predicate | Usage |
|---|---|
| eq | Equal to |
| neq | Not equal to |
| gt | Greater than |
| gte | Greater than or equal to |
| lt | Less than |
| lte | Less than or equal to |
| between | Between two values excluding the upper bound |

You may be wondering: what if we wanted to order the output of Example 4-7?

If you wanted to return the results in a decreasing order, you would do that by adding in the `order().by()` pattern, shown on lines 10 and 11 in Example 4-8.

*Example 4-8.*

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         out("uses").
3         in("charge").
4         has("timestamp",
5             between("2020-12-01T00:00:00Z",
6                     "2021-01-01T00:00:00Z")).
7         out("pay").
8         groupCount().
9           by("vendor_name").
```

2 Kelvin Lawrence, *Practical Gremlin: An Apache TinkerPop Tutorial*, January 6, 2020, *https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html*.

```
10        order(local).          // Order the map object
11          by(values, desc)     // according to the groupCount map's values
```

The results are now:

```
{
  "Target": "3",
  "Nike": "2",
  "Amazon": "1"
}
```

We threw in the use of scope in a traversal at line 10 with the step `order(local)`.

*Scope*
: Scope determines whether the particular operation is to be performed to the current object (local) at that step or to the entire stream of objects up to that step (global).

For a visual explanation of scope in a traversal, consider Figure 4-11.

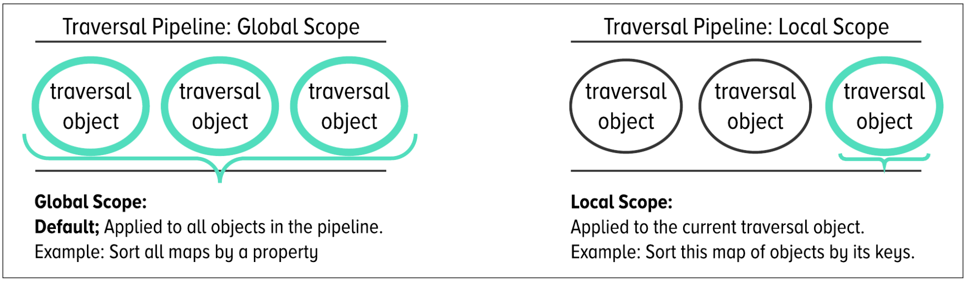



*Figure 4-11. A visual example of the difference between global and local scope in a Gremlin traversal*

To explain it simply, at the end of line 9, we needed to order the object in the pipeline, which is a map. The use of `local` on line 10 tells the traversal to sort and order the items within the map object. Another way to think about this is that we want to order the entries *within* the map. We do that by indicating that the scope is local to the object itself.

The best way to understand traversal scope is to play with different queries in your Studio Notebook and see how the scope affects the shape of your results. More great visual diagrams on understanding the flow of data and object types are available on the DataStax Graph documentation pages.

If you ever question what object type you have in the middle of developing a Gremlin traversal, add `.next().getClass()` to where you are in your traversal development. This will inspect the objects at this point in your traversal and give you their class.

**Query 3: Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.**

The advantage of using a graph database really starts to show as we walk through multiple neighborhoods of data, as we will be doing with this third and last query. Here, we are accessing and mutating data across five neighborhoods of data in our graph. We are going to break this query down into three steps: access, mutation, and then validation.

The first simplification we are going to make to our account is to reduce the scope of the query. We know that Jamie and Aaliyah share only one account: `acct_0`. Therefore, to further simplify our query, we can focus on walking from only one person; we choose Aaliyah.

This brings us to the first shorter query we want to build:

**Query 3a: Find Aaliyah's transactions that are loan payments.** Before we can update important transactions, we need to find the important ones. The transactions we are looking for are those that indicate a loan payment from Aaliyah's joint account to Jamie and Aaliyah's mortgage. Let's outline our approach in pseudocode in Example 4-9 to think about how we are going to walk through our data to answer the question.

*Example 4-9.*

```
Question: Find Aaliyah's transactions that are loan payments
Process:
    Start at Aaliyah's customer vertex
    Walk to her account
    Walk to transactions that are withdrawals from the account
    Go to the loan vertices
    Group and count the loan vertices
```

We used the process outlined in Example 4-9 to create the Gremlin query in Example 4-10.

*Example 4-10.*

```
dev.V().has("Customer", "customer_id", "customer_4"). // accessing Aaliyah's vertex
        out("owns").                        // walking to the account
        in("withdraw_from").                // only consider withdraws
        out("pay").                         // walking out to loans or vendors
        hasLabel("Loan").                   // limiting to only loan vertices
        groupCount().                       // groupCount the loan vertices
          by("loan_id")                     // by their loan_id
```

The results for the sample data will look like:

```
{
  "loan80": "24",
  "loan18": "24"
}
```

Let's step through Example 4-10. On line 1, we start by accessing the customer and walking to their account. On line 2, we traverse to Aaliyah's account. Recalling the schema, we walk through the incoming edge `withdraw_from` to access account withdrawals on line 3.

On line 4, we walk through the `pay` edge label to arrive at either `Loan` or `Vendor` vertices. The `hasLabel("Loan")` step on line 5 is a filter that eliminates all vertices at this point that are not loans. This means we are now considering only the assets into which a payment has been made from the account *and* that are loans. On line 6, we group and count those loan vertices according to their unique identifier, as indicated on line 7.

The result payload indicates that this account has made 24 payments into each loan within the system.

Next, we want to go a step further and update the data in this traversal to indicate which transactions are mortgage payments.

#### Query 3b: Find and update the transactions that Jamie and Aaliyah most value: their payments from their checking account to their mortgage, loan_18.

The traversal required to accomplish this query is a mutating traversal. All we mean by *mutating traversal* is that it updates data in the graph as a part of the traversal. Example 4-11 shows how we can use the traversal above to write properties on the transactions that go from the account and into `loan_18`, because `loan_18` is Jamie and Aaliyah's mortgage loan.

*Example 4-11.*

```
dev.V().has("Customer", "customer_id", "customer_4"). // accessing Aaliyah's vertex
        out("owns").                                  // walking to the account
        in("withdraw_from").                          // only consider withdraws
        filter(
               out("pay").                            // walking to loans or vendors
               has("Loan", "loan_id", "loan_18")).    // only keep loan_18
        property("transaction_type",  // mutating step: set the "transaction_type"
                 "mortgage_payment"). // to "mortgage_payment"
        values("transaction_id", "transaction_type")  // return transaction & type
```

The results are:

```
"144", "mortgage_payment",
"153", "mortgage_payment",
"132", "mortgage_payment",
...
```

Example 4-11 starts the same as the first part of our query. The new portion of this traversal spans lines 4 through 6 with the `filter(out("pay").has("Loan", "loan_id", "loan_18"))` steps. Here, we allow only the transactions that are connected to the `loan_18` vertex to continue down the pipeline. This is because `loan_18` is Jamie and Aaliyah's mortgage loan. On line 7, we mutate the transaction vertices by changing "transaction_type" to "mortgage_payment." At the end of this traversal on line 9, we want to return the `transaction_id` along with its new property, its `transaction_type`.

**Query 3c: Verify that we didn't update every transaction.** At this point, it is very helpful to make sure that we did not update all of Aaliyah's transactions with `mortgage_payment`. We can do that with a quick check, shown in Example 4-12.

*Example 4-12.*

```
// check that we didn't update every transaction
1 dev.V().has("Customer", "customer_id", "customer_4"). // at the customer vertex
2         out("owns").                 // at the account vertex
3         in("withdraw_from").         // at all withdrawals
4         groupCount().                // group and count the vertices
5           by("transaction_type")     // according to their transaction_type
```

The results from the Studio Notebook are shown below. We set `unknown` as the default value during the data loading process also shown in the Studio Notebook:

```
{
  "mortgage_payment": "24",
  "unknown": "47"
}
```

This query does a quick check to validate that we properly mutated our data. Combining lines 1 through 3, we process all of the transactions from Aaliyah's bank account. At line 4, we do a `groupCount()` for all of those vertices according to the value stored in the `transaction_type` property. Here, we see that we correctly updated only the 24 transactions that are mortgage payments to `loan_18`. This validates that our mutation query properly updated our graph structure.

This section started out with three questions, and the last three examples answered them using the Gremlin query language.

We stepped through the basic queries to show you where to start. Get your basic graph walks ironed out before you start exploring the full flexibility and expressivity of the Gremlin query language. We always recommend iterating through Gremlin steps in development mode to find the basic walks that accomplish your queries. This means we are asking you to execute line 1 of a Gremlin query and look at the results. Then execute lines 1 and 2 and look at the results, and so on.

After you have mapped out your basic walks, you can try out more advanced Gremlin. At this point in development, it is very common to find ways to create specific payload structures to pass back to your endpoint.

We will cover the most popular strategies for building JSON with Gremlin in the next section.

## Advanced Gremlin: Shaping Your Query Results

The goal of this section is to build up a more advanced version of our Gremlin query that answers a new question:

```
Is there anyone else who shares accounts, loans, or credit cards with Michael?
```

We would like to introduce a new question to demonstrate advanced Gremlin concepts within a small neighborhood of data. Once you understand how these concepts apply to this question, we invite you to use the accompanying notebook for this chapter to implement the concepts for the other queries introduced in "Basic Gremlin Navigation" on page 97.

We will work through shaping the results of our new query in a few stages. They are:

1. Shaping query results with the `project()`, `fold()`, and `unfold()` steps
2. Removing data from the results with the `where(neq())` pattern
3. Planning for robust result payloads with the `coalesce()` step

> For anyone diving deeper into the world of Gremlin queries, we highly recommend the detail and explanations in the book *Practical Gremlin: An Apache TinkerPop Tutorial* by Kelvin Lawrence.[3]

[3] Kelvin Lawrence, *Practical Gremlin: An Apache TinkerPop Tutorial*, January 6, 2020, *https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html*.

## Shaping Query Results with the project(), fold(), and unfold() Steps

When we start writing a new query, we like to slowly build up its required pieces. One of the most useful Gremlin steps is the `project()` step, because it helps us build up a specific map of data from our query. Let's start building our query out by defining the three keys we want to have in our map: `CreditCardUsers`, `AccountOwners`, and `LoanOwners`.

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3           by(constant("name or no owner for credit cards")).
4           by(constant("name or no owner for accounts")).
5           by(constant("name or no owner for loans"))
```

This query structure is the base of what we are building toward. We want to start with a specific person in this example—namely Michael. Then we want to create a data structure that will have three keys: `CreditCardUsers`, `AccountOwners`, and `LoanOwners`. We create this map with the `project()` step on line 2. The arguments to the `project()` step are the three keys. For each key in the `project()` step, we want to have a `by()` step. Each `by()` modulator creates the values associated to the keys:

1. The `by()` modulator on line 3 will create a value for the `CreditCardUsers` key.
2. The `by()` modulator on line 4 will create a value for the `AccountOwners` key.
3. The `by()` modulator on line 5 will create a value for the `LoanOwners` key.

Let's take a look at the results at this point:

```
{
  "CreditCardUsers": "name or no owner for credit cards",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

This is a good baseline to work from. Next, let's walk through our graph structure to start to populate the values in our map. We will start with the data for the first key: finding people who share a credit card with Michael.

Thinking back to our schema, we will need to walk through the `uses` edge to get to the credit cards. Then we will walk back through the `uses` edge to get back to people. After that, we want to access their names. In Gremlin, we would add this walk on lines 3, 4, and 5:

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
```

```
6         by(constant("name or no owner for accounts")).
7         by(constant("name or no owner for loans"))
```

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
6         by(constant("name or no owner for accounts")).
7         by(constant("name or no owner for loans"))
```

The only steps we added were to walk from Michael out to his credit card via the `uses` edge on line 3. Then, on line 4, we walk back to all people who use that credit card. The resulting payload is:

```
{
  "CreditCardUsers": "Michael",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

This confirms what we know: Michael didn't share any credit cards with other people. We expected to see his name in the result set.

Now let's do the same thing for the next key in our map: `AccountOwners`. Here, we want to walk out the `owns` edge to the account vertex and back to the person vertex:

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2           project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
6         by(out("owns").
7            in("owns").
8            values("name")).
9         by(constant("name or no owner for loans"))
```

Let's look at the resulting payload:

```
{
  "CreditCardUsers": "Michael",
  "AccountOwners": "Michael",
  "LoanOwners": "name or no owner for loans"
}
```

Looking at this data, we do not see what we would expect. We expected to see Maria as a resulting value for `AccountOwners`. Maria does not show up because Gremlin is lazy; it returns the first result, not all results. We need to add a barrier to force all results to finish and return.

The barrier that we like to use here is `fold()`. The `fold()` step will wait for all of the data to be found and then roll up the results into a list. This is a bonus, because now we can build up specific data type rules for our application. The adjusted query reads:

```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(constant("name or no owner for loans"))
```

The shape of the data in the resulting payload is what we were expecting to see:


```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
    "Michael",
    "Maria"
  ],
  "LoanOwners": "name or no owner for loans"
}
```


Let's complete the construction of our map by adding in the statements in the last `by()` step. These statements need to walk from Michael out to his loan and then back. The query and result set are:

```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(out("owes").
           in("owes").
           values("name").
           fold())
```


```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
```

```
      "Michael",
      "Maria"
    ],
    "LoanOwners": [
      "Michael"
    ]
  }
```


```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(out("owes").
           in("owes").
           values("name").
           fold())
```


```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
    "Michael",
    "Maria"
  ],
  "LoanOwners": [
    "Michael"
  ]
}
```


At this point, we have the expected results. We see that Michael shares an account with Maria. And we see that Michael doesn't share credit cards or loans with anyone else.

For some applications, it isn't helpful to return that Michael shares a credit card with himself. Let's dive into how we would remove Michael from this resulting payload.

## Removing Data from the Results with the where(neq()) Pattern

It might be useful for you to eliminate Michael from the result set. We can do that by using the `as()` step to store Michael's vertex, and then eliminate it from the result set. You can remove a vertex from your pipeline with the step `where(neq("some_stored_value"))`.

The next version of our query, in which we have directly applied this step to each section, is shown in Example 4-13.

*Example 4-13.*

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold()).
8         by(out("owns").
9            in("owns").
10             where(neq("michael")).
11           values("name").
12           fold()).
13        by(out("owes").
14           in("owes").
15             where(neq("michael")).
16           values("name").
17           fold())
```

The full results of Example 4-13 are shown below:

```
{
  "CreditCardUsers": [],
  "AccountOwners": [
    "Maria"
  ],
  "LoanOwners": []
}
```

The main additions to our query occur on lines 1, 5, 10, and 15 in the above query. On line 1, we store the vertex for Michael with the `as("michael")` step. Let's take a look at what is happening with `where(neq("michael"))` on line 5, which is the same thing that is happening on lines 10 and 15.

To understand what is happening on line 5, you need to remember where you are in your graph. At the end of line 4, we are on `Customer` vertices. Specifically, we are processing customers that share an account with `Michael`. This is where the `where(neq("michael"))` step comes in. We want to apply a true/false filter to every vertex in the pipeline. The true/false filter test is whether or not that vertex is equal to Michael: `where(neq("michael"))`. If the vertex is Michael, line 5 eliminates it from the traversal. If the vertex is not Michael, the vertex passes through the filter and remains in the pipeline.

## Planning for Robust Result Payloads with the coalesce() Step

Depending on your team's data structure rules, checking whether or not a value in your data payload is an empty list may not be preferred. We can help design around that.

We can implement try/catch logic so that your query doesn't return an empty list. We will step through this for the first key in the map: `CreditCardUsers`. After we step through that, we will add in the full query details for the two remaining `by()` steps.

Let's rewind and go back to just building up the JSON payload for the value associated to `CreditCardUsers`. We are starting from here:

```
dev.V().has("Customer", "customer_id", "customer_0").as("michael").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
             where(neq("michael")).
           values("name").
           fold()).
        by(constant("name or no owner for accounts")).
        by(constant("name or no owner for loans"))
```


```
{
  "CreditCardUsers": [],
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```


You can implement try/catch logic in Gremlin with the `coalesce()` step. We want to shape the results so that there is always a value in the lists for each key, like `"CreditCardUsers": ["NoOtherUsers"]`. Let's start by seeing how to integrate the `coalesce` step into our query:

```
dev.V().has("Customer", "customer_id", "customer_0").as("michael").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
             where(neq("michael")).
           values("name").
           fold().
           coalesce(constant("tryBlockLogic"),    // try block
                    constant("catchBlockLogic"))).// catch block
        by(constant("name or no owner for accounts")).
        by(constant("name or no owner for loans"))
```

The resulting payload is:


```
{
  "CreditCardUsers": "tryBlockLogic",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```


When you use the `coalesce()` step in line 8, it takes two arguments. The first argument is on line 8 and can be thought of as the try block logic. The second argument is on line 9 and can be thought of as the catch block logic.

If the try block logic succeeds, then the resulting data is passed down the pipeline. In this case, for illustrative purposes, we used something that would definitely succeed: the `constant()` step. This step returned the string `"tryBlockLogic"` that we see in the resulting payload. The `constant()` step is useful for many reasons, one of which is that it can serve as a placeholder while you build up more complicated queries. This is how we are using it here.

Should the first argument of the `coalesce()` step fail on line 8, the second argument will execute on line 9. Let's look at how we can use this to populate what we want in our data payload:

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold().
8            coalesce(unfold(),                   // try block
9                     constant("NoOtherUsers"))). // catch block
10        by(constant("name or no owner for accounts")).
11        by(constant("name or no owner for loans"))
```


```
{
  "CreditCardUsers": "NoOtherUsers",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```


On line 8, the logic that we added to the try block is the `unfold()`. This is trying to take the results from the previous step and successfully unfold them. The results at this point in the pipeline are an empty list `[]`. In Gremlin, you cannot unfold an empty object. This throws an exception that is caught by the try block. Therefore, we execute line 9, the second argument of the `coalesce()` step: `constant("NoOtherUsers")`. This is why we see the entry `"CreditCardUsers": "NoOtherUsers"` in our result payload.

Regrettably, we lost our guaranteed list structure. We can add that back in with a `fold()` after the `coalesce()` step:

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold().
8            coalesce(unfold(),
9                     constant("NoOtherUsers")).fold()).
10        by(constant("name or no owner for accounts")).
11        by(constant("name or no owner for loans"))
```

```
{
  "CreditCardUsers": [
    "NoOtherUsers"
  ],
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

The steps we added from line 5 to line 9 create a predictable data structure to exchange throughout your application. It will be well-formatted JSON about which other applications can reason.

Next, we need to add this try/catch logic to each `by()` step. The full logic pattern to add at the end of each `by()` step in our full query is:

```
coalesce(unfold(),                    // try to unfold the names
         constant("NoOtherUsers")). // inject this string if there are no names
fold()                                // structure the results into a list
```

This Gremlin pattern ensures we have a nonempty list in the resulting payload. The full query and its results are:

```
dev.V().has("Customer", "customer_id", "customer_0").as("michael").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
             where(neq("michael")).
           values("name").
           fold().
           coalesce(unfold(),
                    constant("NoOtherUsers")).fold()).
         by(out("owns").
           in("owns").
             where(neq("michael")).
           values("name").
           fold().
           coalesce(unfold(),
                    constant("NoOtherUsers")).fold()).
```

```
        by(out("owes").
           in("owes").
             where(neq("michael")).
           values("name").
           fold().
           coalesce(unfold(),
                    constant("NoOtherUsers")).fold())
```

```
{
  "CreditCardUsers": [
    "NoOtherUsers"
  ],
  "AccountOwners": [
    "Maria"
  ],
  "LoanOwners": [
    "NoOtherUsers"
  ]
}
```

We find that iterative building and stepping through Gremlin steps is the best way to wrap your head around the query language. This book is about teaching you our thought processes, and this is how we think through using Gremlin. There is more than one way to write a graph query; we hope you are curious about using other steps to process the same data. Figuring this out can be as easy as opening up a Studio Notebook and exploring new steps on your own.

# Moving from Development into Production

Bringing back our scuba analogy from the beginning of this chapter, our time training in the pool has come to a close. As we see it, the progression through the technical examples in this chapter is just like learning buoyancy control or deepwater troubleshooting within a pool. At some point, you have learned everything you can from practicing in a controlled environment.

With the foundation we have built over the past few chapters, it is time to take the leap out of your development environment and build a production-ready graph database.

Before you get too concerned, this doesn't mean you are supposed to know everything there is to know about graph data. There are still myriad topics we are continuing to explore ourselves.

What it does mean, however, is that we think you are ready to move into a deeper understanding of using graph data in distributed systems. We set up this example to get you ready for one last step down into the physical data layer of understanding graph data structures in Apache Cassandra. Specifically, the upcoming chapter will show you how to optimize your graph structures for distributed applications.

While illustrating how we think through graph data, we purposefully set up some traps in the example in this chapter. In the next chapter, we will show these traps to you and walk you through their resolution. This upcoming chapter will be the last chapter that uses our C360 example, as it will describe the final iteration in creating a production-quality graph schema for this example.

CHAPTER 5

# Exploring Neighborhoods in Production

When you use DataStax Graph, you are working with graph data in Cassandra. And if you have been following along and executing the implementation details from the last two chapters, you have already been using it.

The paradigm shift from working with a traditional database to working with Apache Cassandra is that we write our data according to how we are going to read it.

To illustrate how we apply this, the examples in Chapters 3 and 4 used but skipped over fundamental topics of working with graph data in Apache Cassandra. Concepts like edge direction and partition key design are fundamental to building a production-quality, scalable, and distributed graph data model.

We are going to dig deeply into the topics of distributed data to set you up for a successful use of distributed graph technology within your production stack.

Recall that we mentioned at the end of Chapter 4 that we purposely set up some traps. Our example built up the schema shown in Figure 5-1 and aimed to use queries like we have in Example 5-1.

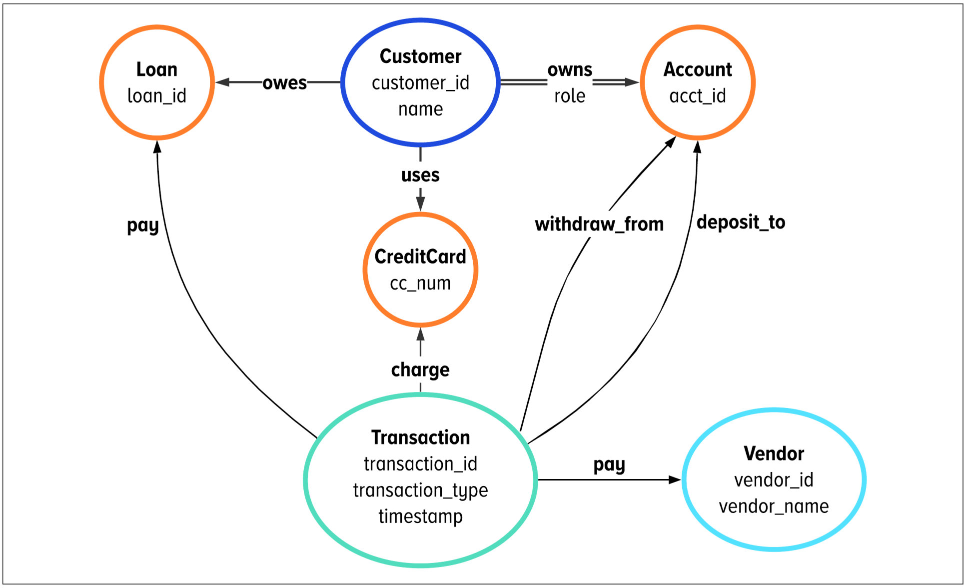



*Figure 5-1. The developmental data model for a graph-based implementation of a C360 application from the previous chapter*

We need to connect two concepts together so you can see the whole picture. First, all of our queries have used the development traversal source `dev.V()`. The development traversal source in DataStax Graph enables you to walk around your data without worrying about indexing strategies. Second, our queries walk from an account vertex to transactions. The query in Example 5-1 uses the production traversal source `g.V()`. If you try to run the query in Example 5-1 in DataStax Studio, you will see something like the execution error in Example 5-1.

*Example 5-1.*

```
g.V().has("Customer", "customer_id", "customer_0"). // the customer
      out("owns").                                  // walk to their account(s)
      in("withdraw_from", "deposit_to")             // access all transactions
```

*Table 5-1. An example of an execution error due to trying to walk an edge in the reverse direction without an index*

| Execution error |
| --- |
| com.datastax.bdp.graphv2.engine.UnsupportedTraversalException: |
| One or more indexes are required to execute the traversal |

This error is tied to the representation of graph data structures on disk. In the rest of this chapter, we take a peek under the hood to explain the *why* and then apply the *how*.

## Chapter Preview: Understanding Distributed Graph Data in Apache Cassandra

The primary intent of this chapter is to introduce design and operational recommendations for modeling data efficiently prior to entering production. For that, this chapter builds on the example from Chapter 4 by detailing how graph data structures operate in Apache Cassandra.

> At the end of this chapter, you will have a list of 10 data modeling recommendations to apply to any new problem. We will use these same tips throughout the remaining examples in this book, too.

We selected the next set of technical topics to illustrate the minimum required set of concepts for building production-quality distributed graph applications. This chapter has three main sections that align with the accompanying notebook and technical materials.

The first section of this chapter revisits the topics we used but did not explain in Chapter 4. Here, we introduce the fundamentals of distributed graph structures to model our queries from that chapter. Namely, you will learn about partition keys, clustering columns, and materialized views.

The second section applies the concepts of distributed graph structures to our second set of data modeling recommendations. We will introduce Cassandra topics such as denormalization, revisit edge direction, and talk about loading strategies. These tips represent data modeling decisions that we recommend for production-quality, distributed graph schema.

The last section walks through the final iteration of our C360 example. We will explain the schema code that applies the concepts of materialized views and indexing strategies. And we will go through one last iteration of our Gremlin queries to use the new optimizations.

Altogether, the thought process and development in Chapters 3, 4, and 5 represent the development life cycle of designing, exploring, and finalizing the models and queries for your first application with distributed graph data.

Let's get started by taking a final step down into the physical data layer of working with graph data in Apache Cassandra.

## Working with Graph Data in Apache Cassandra

This section looks at the fundamental concepts of working with graph data structures in Apache Cassandra: primary keys, partition keys, clustering columns, and materialized views.

We are going to discuss these Cassandra data modeling topics from a graph user's perspective.

First, we will talk about what you need to know about vertices, and then we will go over what you need to know about edges. For vertices, you need to know about primary keys and partition keys. For edges, you need to know about clustering columns and materialized views.

Let's get started with the concept that connects everything: the primary key.

### The Most Important Topic to Understand About Data Modeling: Primary Keys

A major challenge of building a good data model within a distributed system is determining how to uniquely identify your data with primary keys.

You have already worked with one of the simplest forms of a primary key: the partition key.

*Partition key*
: The partition key is the first element of a primary key in Apache Cassandra. The partition key is the part of the primary key that identifies the location of the data in a distributed environment.

From a user's perspective, the entire primary key is required for you to access your data from the system. The partition key is just the first piece of the primary key.

*Primary key*
: The primary key describes a unique piece of data in the system. In DataStax Graph, a primary key can be made up of one or more properties.

You have already been using and working with primary and partition keys. In DataStax Graph, you specify the desired primary key in the schema API. We saw the simplest version of a primary key—just one partition key—in the previous chapter with:

```
schema.vertexLabel("Customer").
       ifNotExists().
       partitionBy("customer_id", Text). // basic primary key: one partition key
       property("name", Text).
       create();
```

The partitionBy() method indicates the value that will be included in the label's partition key. In this case, we have only one value, customer_id. This means that customer_id is the full primary key and partition key for the Customer vertex.

From a developer's perspective, this decision has three consequences for your application. First, the value for customer_id uniquely identifies the data. Second, your application will need the value for customer_id to read the data about the customer. We will cover the third point in a moment.

These two consequences govern how you, the user, design your data's primary and partition keys. Let's take a look at an example. Previously, you used your primary key to look up this data in Gremlin via:

```
g.V().has("Customer", "customer_id", "customer_0").
      elementMap()
```

This returns:

```
{
  "id": "dseg:/Customer/customer_0",
  "label": "Customer",
  "name": "Michael",
  "customer_id": "customer_0"
}
```

Looking up vertices or edges by their full primary key is the fastest way to read data in DataStax Graph. This is one of the main reasons that selecting a good partition and primary key for your data is so important.

There is a third consequence of the partition key in Apache Cassandra. A vertex label's partition key assigns your graph data to a host within a distributed environment. Partition keys also give you different ways you can colocate your graph data. Let's dig into the details.

## Partition Keys and Data Locality in a Distributed Environment

We recommend this section if you like getting deep in the weeds.

This section aims to synthesize topics across the Cassandra and graph communities. We will explore some hypothetical alternatives to graph partitioning by examining different partition key choices to colocate graph data. We will conclude with the partition strategy we started with for our example, but you will have gained a better understanding of the effects of partition key design and the graph partitioning problem.

And we need to be pedantic about what we really mean for a brief moment.

The word *partition* means two very different things to two different groups of people. The Cassandra community's understanding of *partition* answers the question,

"Where is my data in my cluster?" The graph community's understanding of the term answers the question, "How can I organize my graph data into a smaller group to minimize an invariant?"

This book applies the Cassandra community's definition of partitioning to working with graph data. When we refer to a partition, we are referencing data locality, or on which server your data is written to disk across your distributed system.

To illustrate how we will be using the idea of partitioning, let's recall some data for our current example, as shown in Figure 5-2.

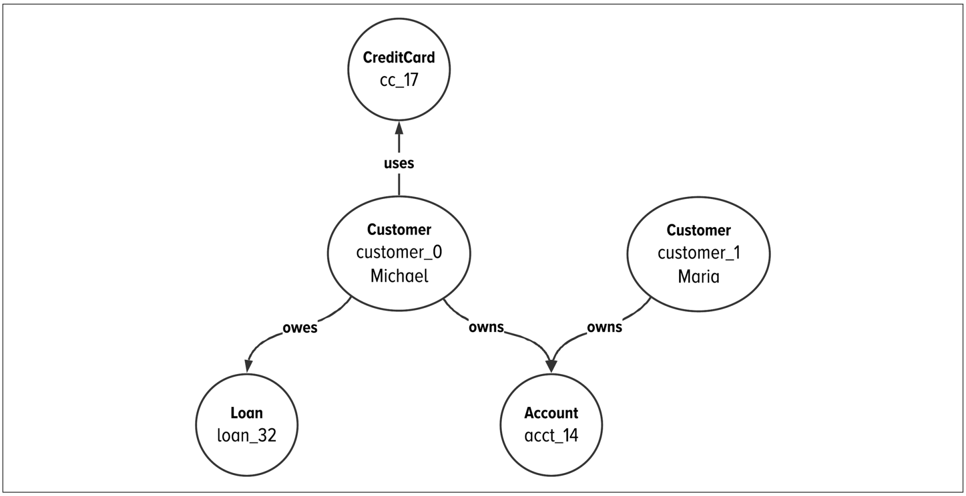



*Figure 5-2. Sample data for three customers in our C360 example*

To visualize data assignment to a server (also referred to as an instance or a node) in a cluster, imagine you are working with a cluster of four servers running DataStax Graph in Apache Cassandra. In Figure 5-3, we represent a distributed cluster with a circle that has four servers running Cassandra. (Each eye in Figure 5-3 represents DataStax Graph in Cassandra.) Then, we show where your graph data is written to disk by illustrating the graph data next to the server around your cluster, as we do in Figure 5-3.

The largest circle in Figure 5-3 represents a cluster of four servers, each indicated with the Cassandra eye logo, running DataStax Graph in Apache Cassandra. The sample data from Figure 5-2 is shown next to the server in which the data is physically stored. In Apache Cassandra, data is mapped to a specific server in your cluster according to its partition key.

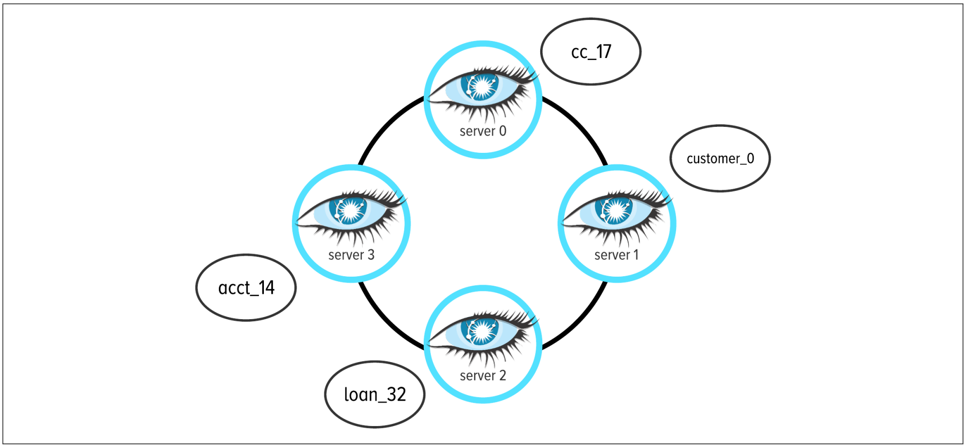



*Figure 5-3. Illustrating which server (node) each vertex is stored in a distributed cluster; the circle with four eyes represents a distributed cluster*

In Figure 5-3, you see that the data for `customer_0` is mapped to four different machines. The customer vertex is written to server 1, the loan vertex is on server 2, the account vertex is written to machine 3, and the credit card vertex is on machine 0.

You can think of partition keys and their association to data locality in a distributed environment as follows: data with the same partition key is stored on the same machine, and data with different keys may be stored on different machines.

#### Partitioning graph data according to access pattern

With graph data, there are strategies for designing your partition keys to minimize the latency of your graph traversals. Different partitioning strategies affect the colocation of your data and, therefore, the latency of your query.

To minimize jumping around machines in your cluster when you are processing your graph data, you may consider a partitioning strategy that keeps all the data related to your query within the same partition. To illustrate this idea, Figure 5-4 shows a partitioning strategy optimized for the expected access pattern of a C360 application. The partitions are defined according to the individual customer and their data because a C360 query will typically be looking for an individual and their associated data. For our sample data, we would create a partition for each individual.

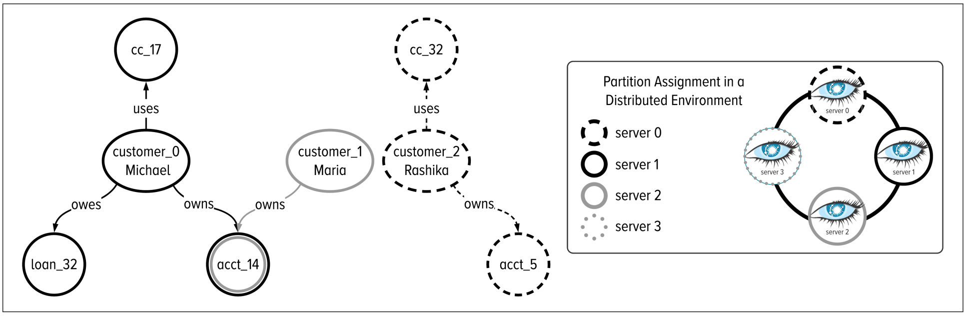



*Figure 5-4. Partitioning graph data in a distributed system according to access pattern*

> If you have a background in graph theory, the partitioning strategy illustrated in Figure 5-4 is similar to partitioning according to connected components.
>
> If you have a background in working with Apache Cassandra, the partitioning strategy illustrated in Figure 5-4 follows the same practice of partitioning according to access pattern.

To implement the partitioning strategy illustrated in Figure 5-4, you would need to add the customer's unique identifier as the partition key for every vertex label. In your schema code, we implement the partitioning strategy with:

```
schema.vertexLabel("Account").
       ifNotExists().
       partitionBy("customer_id", Text).
       clusterBy("acct_id", Text).  // to be defined in a coming section
       property("name", Text).
       create();
```

It is useful to consider this partitioning strategy because it minimizes the latency of your query. All the data for your customer-centric query is colocated on the same node in your environment. This is an optimization at the physical data layer that will be advantageous when you query your data.

However, there are two reasons why this type of partition strategy will not be recommended for the queries we are exploring for our example. Recall that the full primary key is required to start your graph query. The first downside to the design shown in Figure 5-4 is that you will need to know the customer's identifier to start your graph traversal at an account.

Applying this to our example of a shared account, `acct_14`, brings us to another drawback to using this partition strategy. This schema design will create two vertices about `acct_14` that are adjacent to two different people. This means that you won't be able to start at `acct_14` and find all customers who own that account. This has implications for your graph query.

For the C360 queries we are exploring in this example, the partition strategy from Figure 5-4 doesn't make sense. When we talk about trees in an upcoming example, however, it makes sense to consider data model optimizations to minimize query latency.

#### Partitioning according to unique key

Let's look at a second strategy and compare it to colocating data according to your application's access pattern.

Think back to the full schema for our example and recall that each vertex label had a single, unique partition key. You can think of this as separating your graph data via the most granular division possible: the data's most unique value.

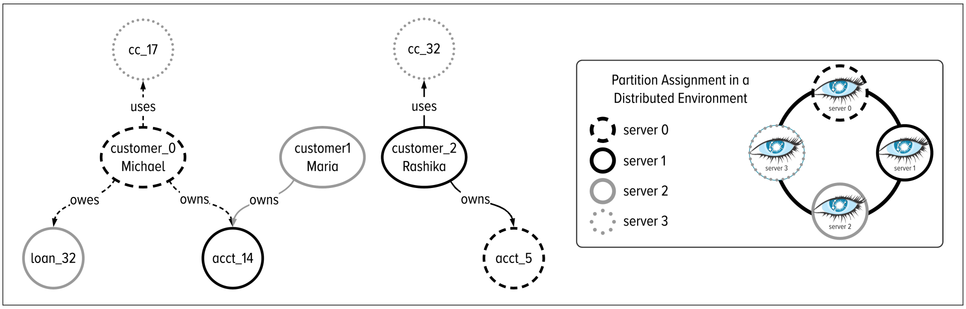



*Figure 5-5. A visual example of the different partitions within your graph data according to the vertex label's partition key*

The graph data in Figure 5-5 would distribute the vertices across your cluster according to the partition key's value. Essentially, each vertex will be mapped to a different partition because each partition key's value is unique.

One of the drawbacks to partitioning your vertices according to a unique key is that any time you need to walk through your data, you will be jumping between machines across your distributed environment. The purpose of using graph data in your application is to use the connections and relationships in your data. If you structure your graph data across a distributed environment according to unique identifiers, that also means that you will (likely) be switching servers each time you need to access connected data.

#### Final thoughts on partitioning strategies

There are benefits and drawbacks to the different strategies for partitioning graph data in a distributed environment. Partitioning your graph data according to access pattern creates limitations on how you can walk through connected data. On the other hand, this strategy minimizes traversal latency by colocating large components of data onto the same node.

The most common way to partition your graph data is by the data's unique identifier. This makes it easiest to plan for query flexibility but does also introduce latency to your queries due to the distributed nature of your graph. This is the approach we will use for our C360 example.

The only way to understand the implications of any partition strategy is to calculate what it would look like for your data and queries. This requires a balance between understanding the distributions of the data for the application you need to build today and considering the future scope toward which you are building.

> Selecting a good partitioning strategy is more complicated when we are working with graph data. Partitioning graph data around a distributed environment is synonymous with breaking up your graph data into different sections. Optimizing which data belongs to a particular section is classified as one of the hardest types of problems in computer science: an NP-complete problem. While maybe not the best news, this helps to explain why using graph technologies in a distributed environment isn't as simple as translating an entity-relationship diagram into a graph data model.

On the topic of partitioning, there are two main takeaways to restate here: uniqueness and locality. In DataStax Graph, your data's primary key is its unique identifier. For the fastest performance, you start your graph queries by data via its full primary key.

The second thing to note is that your data's partition key determines its locality in your cluster. This governs which machines in your cluster will store the data and the colocation of other data alongside it.

Given uniqueness and locality with partition keys, let's take a look at how edges are represented in Apache Cassandra.

## Understanding Edges, Part 1: Edges in Adjacency Lists

Diving into the world of graph modeling brings a large wave of terms, concepts, and thinking patterns. Now that we have the basics covered, let's take a look at how graph data, namely the edges, can be represented on disk or in memory.

There are three main data structures for representing edges in data:

*Edge list*
: An edge list is a list of pairs in which every pair contains two adjacent vertices. The first element in the pair is the source (from) vertex, and the second element is the destination (to) vertex.

*Adjacency list*

An adjacency list is an object that stores keys and values. Each key is a vertex, and the value is a list of the vertices that are adjacent to the key.

*Adjacency matrix*

An adjacency matrix represents the full graph as a table. There is a row and column for each vertex in the graph. An entry in the matrix indicates whether there is an edge between the vertices represented by the row and column.

To understand these data structures, let's look at how we would map a small example of graph data into each structure.

There is a significant amount of detail illustrated in Figure 5-6. At the top, we show an example of five vertices that are connected by four edges. Direction matters when you map the data into each of the graph data structures below.

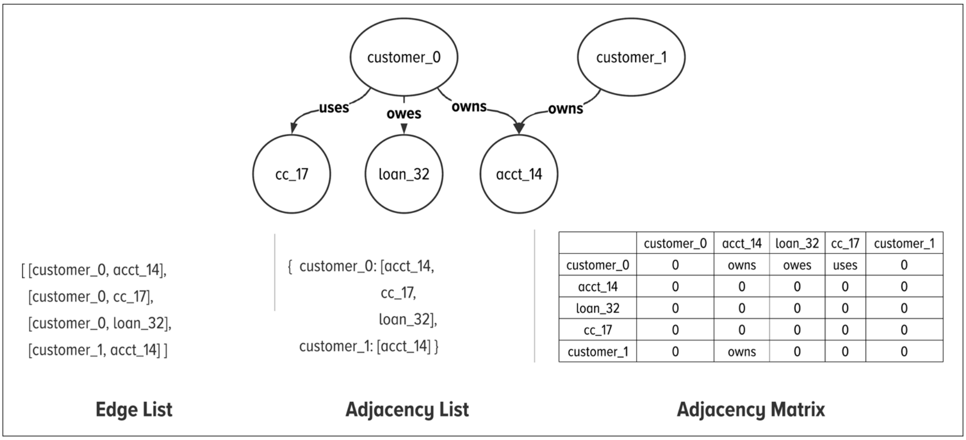



*Figure 5-6. An example of three different data structures for storing edges on disk*

Let's walk through each data structure.

On the lower left in Figure 5-6, we have written out how the example data would be stored in an edge list. The edge list contains four entries: one entry per edge in our example data. In the center, we represent how the example data maps to an adjacency list. The adjacency list has two keys: one key per vertex with outgoing edges. The value for each key is a list of the incoming vertices the edges point to. The last data structure, shown on the far right, is an adjacency matrix. There are five rows and five columns: one row or column for each vertex in the graph. Each entry in the matrix indicates whether there is an edge going from the row vertex to the column vertex.

There are space and time trade-offs for each data structure. Skipping over optimizations that can be made for each individual data structure, let's consider the complexities of each at a basic level. Edge lists are the most compressed version of representing

your graph, but you have to scan the entire data structure to process all edges about a specific vertex. Adjacency matrices are the fastest way to walk through your data, but they take up an inordinate amount of space. Adjacency lists combine the benefits of the other two models by providing an indexed way to access a vertex and limit the list scans to only the individual vertex's outgoing edges.

In DataStax Graph, we use Apache Cassandra as a distributed adjacency list to store and traverse your graph data. Let's dig into how we optimize the storage of edges on disk so you can get the most benefit out of the sorting order of your edges during your graph traversals.

## Understanding Edges, Part 2: Clustering Columns

You used the concept of clustering columns when you added edge labels to your graph in Chapter 4.

*Clustering column*
: A clustering column determines a sorting order of your data in tables on disk.

Clustering columns make up the final components of a table's primary key in Cassandra. Clustering columns inform the database how to store the rows in a sorted order on disk, which makes data retrieval more efficient.

We want to dig into the details of clustering columns because they explain two concepts at the same time. First, the technical implications of clustering columns detail exactly why the query at the beginning of this chapter returned an error. Second, clustering columns illustrate how we sort your edges on disk in an adjacency list structure to provide the fastest access possible.

Example 5-2 illustrates the use of a clustering column in creating an edge label.

*Example 5-2.*

```
schema.edgeLabel("owns").
       ifNotExists().
       from("Customer").   // the edge label's partition key
       to("Account").      // the edge label's clustering column
       property("role", Text).
       create()
```

Following the example in Example 5-2, we can pick out the partition key and clustering columns for our edge label:

1. The `from(Customer)` step means that the full primary key of the `Customer` vertex will be the partition key for the edge label `owns`: `(customer_id)`.
2. The full primary key for `Account` will be the clustering column for the `owns` edge: `(acct_id)`.

Putting this together, we can lay out Cassandra's table structures alongside the graph schema, as see in Figure 5-7.

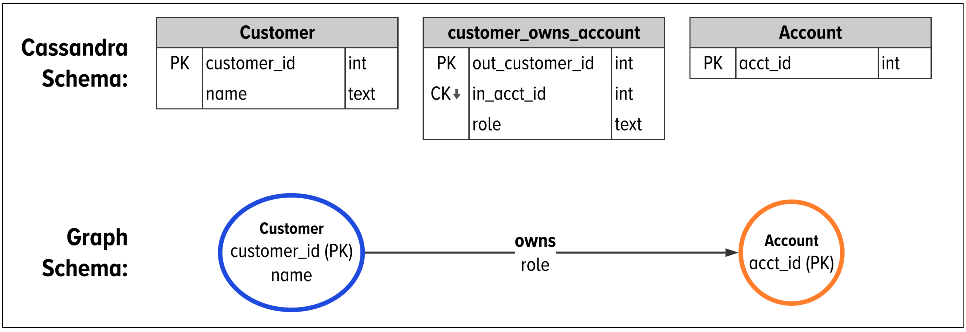



*Figure 5-7. The default table structure in Cassandra for an edge between two vertices*

Figure 5-7 shows the table structures in Cassandra as they map to graph schema using the Graph Schema Language (GSL). The `Customer` vertex creates a table with a partition key, `customer_id`. The `owns` edge connects `Customer` to `Account`. The partition key of the `owns` edge is the `customer_id`. The `owns` edge also has a clustering key, `in_acct_id`, which is the partition key of the account vertex. There is a third column in the `customer_owns_account` table: `role`. This is a simple property and is not a part of the primary key. As a result, the value for `role` will come from the most recent write of an edge between a customer and an account.

To make this concrete, Figure 5-8 shows an example of data that follows the schema from Figure 5-7.

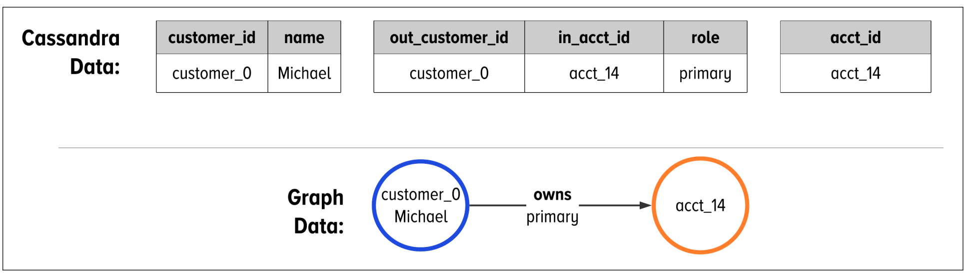



*Figure 5-8. Looking at how our data from the schema in Figure 5-7 would be organized on disk and how it would be represented in DataStax Graph*

Before we move on to a different topic, there is one last idea to synthesize about clustering keys and edges in DataStax Graph. In Chapter 2, we outlined cases in which you want to have many edges between two vertices. We denote this in the GSL with a double-lined edge. In Cassandra, we would make that property a clustering key. Figure 5-9 shows the Cassandra schema alongside a graph schema that models adjacent vertices as a collection.

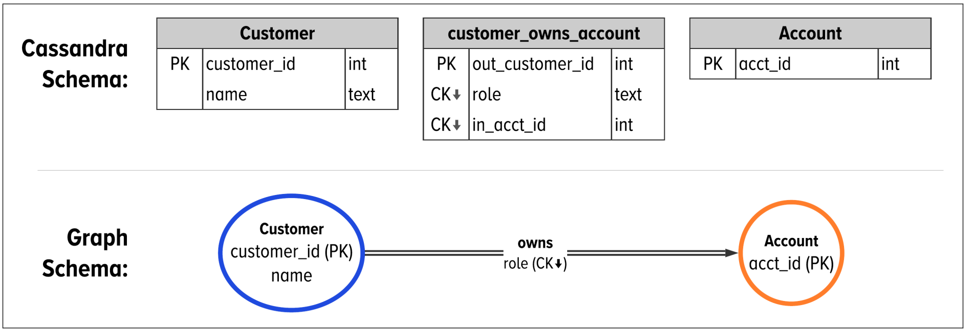



*Figure 5-9. The table structure in Cassandra when we model clustering keys on edges*

Figure 5-10 shows the table structure in Cassandra as they map to graph schema using the GSL when we model the multiplicity of a graph with many edges between instance vertices. The difference is in the table for the `owns` edge. We now have the `role` as a clustering key for this edge, before the clustering key for the `acct_id`. The schema from Figure 5-9 allows there to be multiple edges between vertices, as we show in Figure 5-10.

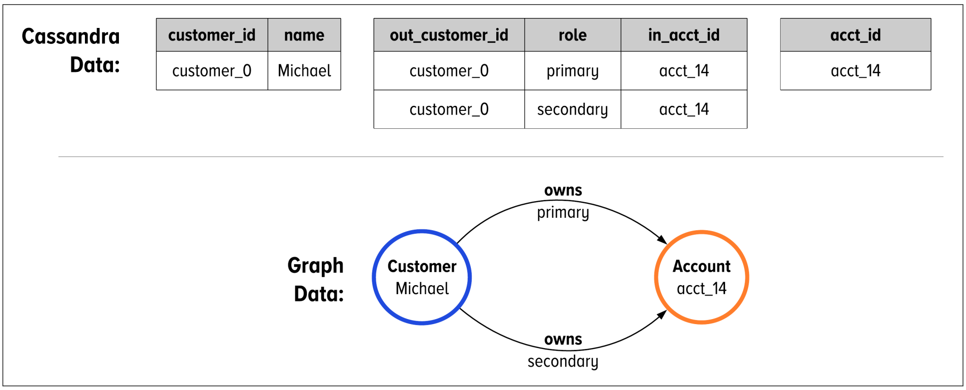



*Figure 5-10. Looking at how our data from the schema in Figure 5-9 would be organized on disk and how it would be represented in DataStax Graph*

Now that we understand the structure of edges on disk, let's visit where they will be stored within a distributed environment.

### Synthesizing concepts: Edge location in a distributed cluster

Recall that the partition key identifies where the data will be written within the cluster. This means that the outgoing edges for a vertex will be stored on the same machine as the vertex itself. We previewed this in Figure 5-5 because the edges have the same color as the customer vertices; they are all orange. To illustrate what we mean, let's look at the locality of edges in our cluster, as shown in Figure 5-11.

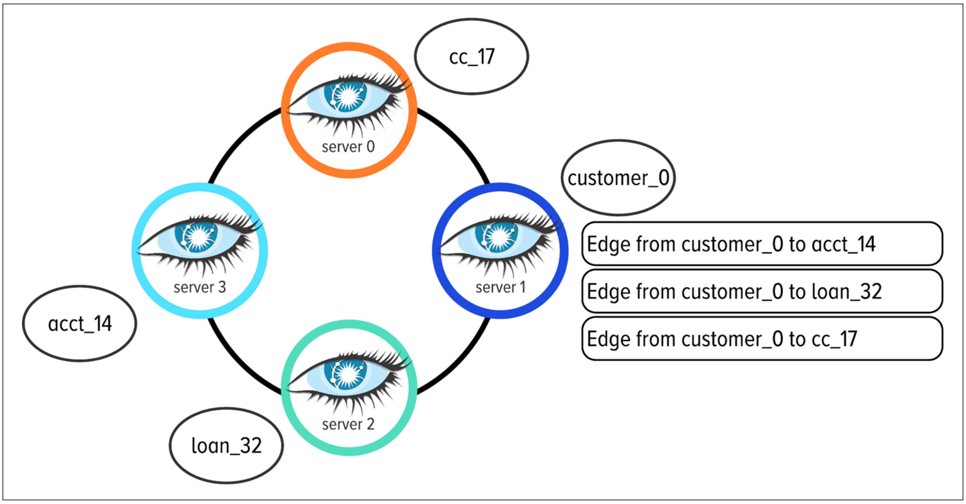



*Figure 5-11. A visual example of data locality for the edges in our example*

The image in Figure 5-11 illustrates where the edges for `customer_0` will be stored within a distributed environment. Each of the edges will be colocated on the same machine as the vertex for `customer_0` because each edge has the same partition key: the `customer_id`.

The next thing to understand is how the edges are sorted within their partition. The full primary key of the *adjacent* vertex label becomes the clustering column(s) of the edge label. This means that the edges are sorted on disk according to their incoming vertex's primary key, as visualized in Figure 5-12.

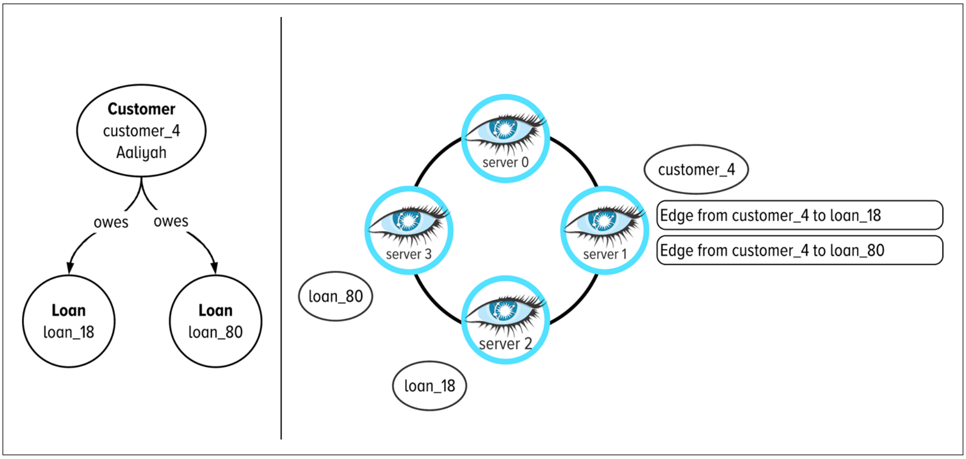



*Figure 5-12. A visual example of mapping graph data, on the left, to its storage location in a distributed cluster, on the right*

The main concept to understand from Figure 5-12 is illustrated on the right. We are showing that the vertex for `customer_4`, Aaliyah, is written to disk on machine 1 in our cluster. Also on machine 1, we will find the outgoing edges from Aaliyah sorted according to their incoming vertex. Aaliyah has two loans connected to her with an `owes` edge. We see that on disk, these edges will be sorted according to the incoming vertex's partition key, the `loan_id`. We see `loan_18` is the first entry and `loan_80` is the second entry.

> To check whether you are synthesizing concepts: where would the customer vertices for Michael, Maria, Rashika, and Jamie be in Figure 5-12? Answer: The partition key for each of those vertices is their `customer_id`, which would be hashed and mapped to any one of the servers. Because we are working with five customers in total, there will be at least one server with two customer vertices. This logic is referred to as the "pigeonhole principle" in mathematics.

You might be asking yourself: why are we getting into all this? It all comes down to the minimum requirement for accessing a piece of data in Apache Cassandra: the partition key.

## Understanding Edges, Part 3: Materialized Views for Traversals

The main area in which you are going to feel the effects of an edge's primary key design comes into how you access your edges. To use an edge, you have to know its partition key.

Because of this, we cannot yet traverse our edges in the reverse direction! This is because there are no edges in the system that start with the partition key from the incoming vertex labels in our examples.

Remember our query from Example 5-1?

```
g.V().has("Customer", "customer_id", "customer_0"). // the customer
      out("owns").                                 // walk to their account(s)
      in("withdraw_from", "deposit_to") // walk to all transactions
```

Recalling the schema we built in the previous chapter, the `deposit_to` edges point from a `Transaction` to an `Account`. However, this query is trying to walk that edge in the reverse direction: from the `Account` to the `Transaction`.

Applying what we just learned about edges in DataStax Graph, we know that this error happens because the edge does not exist on disk. The edge was written from the transaction to the account, but not the reverse.

If we want to walk from accounts to transactions, then we need to store the edge in the other direction as well. This is not done by default in DataStax Graph because of performance implications, similar to how indexing every column in a relational data model is an antipattern.

What we need are bidirectional edges, or edges that go in both directions. This option brings us to the last technical topic in this chapter.

#### Materialized views for bidirectional edges

One of the main reasons engineers love Apache Cassandra is they are willing to trade data duplication for faster data access. This is where materialized views come into play with DataStax Graph. From the user's perspective, you can think of a materialized view as follows:

*Materialized view*
: A materialized view creates and maintains a copy of the data in a separate table with a different primary key structure, rather than requiring your application to manually write the same data multiple times to create the access patterns you need.

Under the hood, DataStax Graph uses materialized views to be able to walk an edge in its reverse direction.

To demonstrate, Example 5-3 shows how to create a materialized view on the existing edge label for `deposit_to`.

*Example 5-3.*

```
schema.edgeLabel("deposit_to").
       from("Transaction").
       to("Account").
       materializedView("Transaction_Account_inv").
       ifNotExists().
       inverse().
       create()
```

Example 5-3 creates a table in Apache Cassandra called "Transaction_Account_inv". The partition key for this table is the acct_id. The clustering column is transaction_id.

The full primary key from Example 5-3 is written as (acct_id, transaction_id). This notation means that the full primary key contains two pieces of data: acct_id and transaction_id. The first value, acct_id, is the partition key, and the second value, transaction_id, is the clustering column.

From the user's perspective, this gives us the ability to walk through the deposit_to edge from accounts to transactions. To convince ourselves of this, let's see the edges that are stored between these two data structures by inspecting the data on disk.

We can inspect the edges on disk for the deposit_to edge label by querying the underlying data structures in Apache Cassandra. There are two tables to inspect. First, let's look at the original table for Transaction_deposit_to_Account; you can do this from DataStax Studio with the following (the results are shown in Table 5-2):

```
select * from "Transaction_deposit_to_Account";
```

*Table 5-2. The data layout from the table Transaction_deposit_to_Account*

| Transaction_transaction_id | Account_acct_id |
|---|---|
| 220 | acct_14 |
| 221 | acct_14 |
| 222 | acct_0 |
| 223 | acct_5 |
| 224 | acct_0 |

The following query shows how to list all edges on disk for the materialized view of the deposit_to edge label, and Table 5-3 displays the results:

```
select * from "Transaction_Account_inv";
```

*Table 5-3. The data layout from the table `Transaction_Account_inv`*

| Account_acct_id | Transaction_transaction_id |
|---|---|
| acct_0 | 222 |
| acct_0 | 224 |
| acct_5 | 223 |
| acct_14 | 220 |
| acct_14 | 221 |

Let's look very closely at the differences between Table 5-2 and Table 5-3. The easiest one to spot is the transaction involving `acct_5`. In Table 5-2, we see that the partition key for this edge is `out_transaction_id`, which is `223`. The clustering column is `in_acct_id`, which is `acct_5`.

Examine how this same edge is stored in Table 5-3, the materialized view of Table 5-2. We can see that the edge's keys are flipped; the partition key for this edge is `in_acct_id`, which is `acct_5`, and the clustering column is `out_transaction_id`, which is `223`. We now have bidirectional edges to use in our example.

### How far down do you want to go?

We just walked through all of the technical explanations for topics in Apache Cassandra that we have planned for this book. Our explanations of the technical concepts are intentionally only a surface-level introduction to the internals of Apache Cassandra, presented from the perspective of a graph application engineer. There is much more to understand about partition keys, clustering columns, materialized views, and more within distributed systems.

We encourage you to go deeper and can recommend two other resources to get you there.

First, for a deep dive on the internals of Apache Cassandra, consider picking up a different O'Reilly book: *Cassandra: The Definitive Guide*, Third Edition by Jeff Carpenter and Eben Hewitt (O'Reilly).

Or for a complete examination of the internals of distributed systems, check out Alex Petrov's *Database Internals: A Deep Dive into How Distributed Data Systems Work* (O'Reilly).

### Where are we going from here?

We are coming back up from the internals of distributed graph data for one last pass of our C360 example. Applying the concepts we have discussed can give us more data modeling recommendations, schema optimizations, and a few new ways to implement our Gremlin queries. So that is exactly where we are going.

The upcoming section applies our knowledge of keys and views in Apache Cassandra to data modeling best practices with DataStax Graph.

# Graph Data Modeling 201

The new knowledge of the layout of vertices and edges in DataStax Graph opens up more data modeling optimizations. Let's apply our understanding of partition keys, clustering columns, and materialized views and visit our second set of data modeling recommendations (picking up from the six recommendations provided in Chapter 4).

To begin with, let's recall where our graph schema left off—see Figure 5-13.

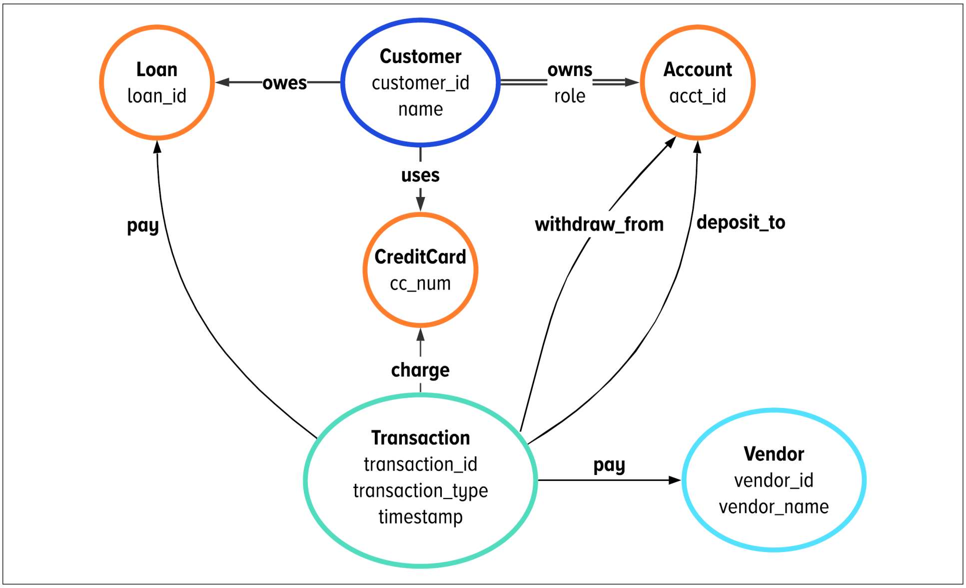



*Figure 5-13. Our development schema from Chapter 4*

This brings us to our next data modeling recommendation.

**Rule of Thumb #7**

Properties can be duplicated onto edges or vertices; use denormalization to reduce the number of elements you have to process in a query.

To apply this tip, consider a case in which an account has thousands of transactions. When we want to find the most recent 20 transactions, we need to access the account vertex by walking through all transactions *before* we can subselect the vertices by time. It is pretty expensive to traverse all of the edges to access all transactions and *then* sort the transaction vertices.

Can we be smarter and reduce the amount of data we have to process?

We can. Specifically, we can store a transaction's time in two places: on the transaction vertex and on the edges. This way, we can subselect the edges to limit our traversal to only the most recent 20 edges. Figure 5-14 illustrates duplicating time onto an edge label.

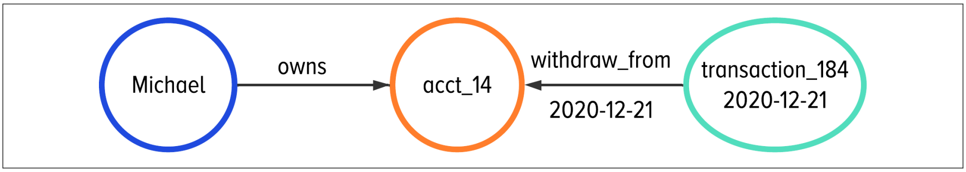



*Figure 5-14. Applying denormalization to include a timestamp on the edges and vertices as an optimization to improve read performance*

For simplicity's sake, in Figure 5-14 we show only the addition of a timestamp to the `withdraw_from` edge; we will apply the same technique for the `deposit_to` and `charge` edge labels.

This type of optimization requires your application to write the same timestamp onto both the edge and the vertex. This is called *denormalization*.

*Denormalization*
: Denormalization is the strategy of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data grouped differently.

Duplicating properties, or denormalization, is a very popular strategy that balances the dualities between unlimited query flexibility and query performance. On one hand, modeling your data in a graph database allows for more flexibility and easier integration of data sources. This flexibility is one of the main reasons teams are picking up graph technologies; graph technology inherently integrates more expressive modeling and query languages.

On the other hand, poor planning during development has left many teams before you with unrealistic expectations for their production graph model. They focused more on data model flexibility at the expense of query performance. Your queries will be more performant if you take advantage of modeling tricks like denormalization.

Before you start adding properties and materialized views to all of your edges, consider our next recommendation.

**Rule of Thumb #8**

Let the direction you want to walk through your edge labels determine the indexes you need on an edge label in your graph schema.

With this tip, we are asking you to do a few things. First, we are advising you to work out your Gremlin queries in development mode first, just like we did in Chapter 4. Then we can apply those final queries to determine *only* the materialized views that you need. You don't need indexes for everything.

There are two ways to do this in DataStax Graph: you can do it yourself, or you can tell the system to do it.

Let's start with what it would look like if you were to figure out indexes on your own.

To recognize when you need an index, you have to map your Gremlin query onto your graph schema. Mapping a query onto schema is something we've been mentally practicing throughout this book, but let's see what this looks like drawn out in Figure 5-15. We will draw out our first query's steps in our schema from start to end. Then, we use the query steps overlaid on our schema to identify where we will need an edge index. Figure 5-15 depicts a query's steps drawn over schema followed by Example 5-4, which shows the Gremlin query.

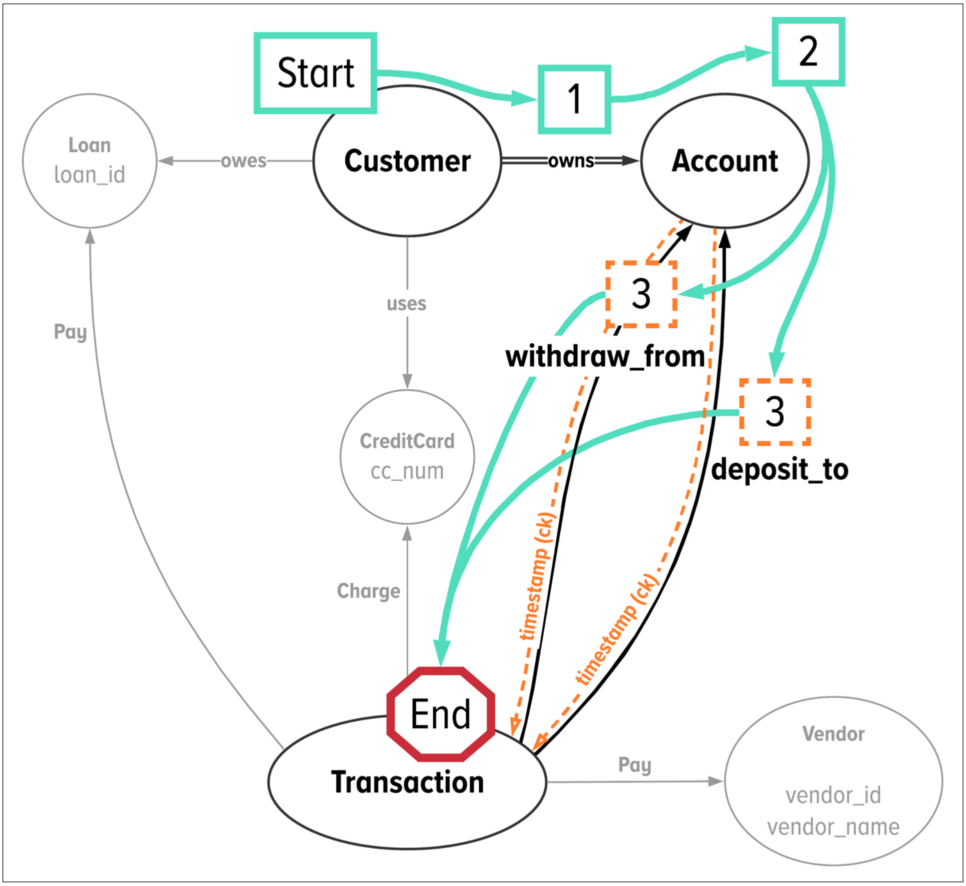



*Figure 5-15. Mapping your query onto your schema to find where you need a materialized view on an edge label*

*Example 5-4.*

```
dev.V().has("Customer", "customer_id", "customer_0").  // [START]
        out("owns").                         // [1 & 2]
        in("withdraw_from", "deposit_to"). // [3]
        order().                             // [3]
          by(values("timestamp"), desc).    // [3]
        limit(20).                           // [3]
        values("transaction_id")             // [END]
```

Let's break down what we are showing in Figure 5-15 alongside Example 5-4. We mapped each step of the query to the schema that you walk through during the query. The boxes labeled from `Start` to `End` map a green path through the schema elements to match the query's steps to where we are walking throughout our schema.

The walk through our schema can be thought of as follows. We begin the traversal by uniquely identifying a customer, shown in the query and schema with the `Start` box. This is line 1 in our query. Then we use the `owns` edge to access that customer's account; this is shown in the boxes labeled `1` and `2`. This is line 2 in our query. Box `3` maps together the processing and sorting of transactions. This maps to lines 3, 4, 5, and 6 in our query. `End` labels where the traversal stops, on line 7 of our query.

The most important concept in Figure 5-15 is at step `3`. The query walks through the incoming `withdraw_from` and `deposit_to` edge labels to access the `Transaction` vertex label. However, we are walking *against* the direction of these edge labels in our schema. We highlighted this in Figure 5-15 with orange dotted lines.

Being able to mentally see that we are walking against the direction of an edge label identifies where you need a materialized view in your graph. This is a very important concept that we hope you followed from Figure 5-15 alongside Example 5-4. We think of this last example as one of the most fundamental aha moments for understanding graph data in Apache Cassandra, and we hope you got there.

## Finding Indexes with an Intelligent Index Recommendation System

If juggling all of this in your head is new or does not feel natural, there is another way: you can let DataStax Graph do it for you.

DataStax Graph has an intelligent index recommendation system called `indexFor`. To let the index analyzer figure out what indexes a particular traversal requires, all you need to do is execute `schema.indexFor(<your_traversal>).analyze()` using the query we walked through in Figure 5-15:

```
schema.indexFor(g.V().has("Customer", "customer_id", "customer_0").
                    out("owns").
                    in("withdraw_from", "deposit_to").
                    order().
                      by(values("timestamp"), desc).
                    limit(20).
                    values("transaction_id")).
       analyze()
```

Because we already created a materialized view for `deposit_to`, this command will output only one recommendation. The output contains the following information, reformatted here to make it easier to read:

```
Traversal requires that the following indexes are created:
schema.edgeLabel("withdraw_from").
    from("Transaction").
    to("Account").
    materializedView("Transaction__withdraw_from__Account_by_Account_acct_id").
    ifNotExists().
    inverse().
    create()
```

Essentially, Figure 5-15 and `indexFor(<your_traversal>).analyze()` are doing the same thing. They are mapping your traversal onto your schema to see where you need a materialized view.

After you develop all of your queries, as we did in Chapter 4, you can use either technique to figure out where you will need indexes in your production schema. The manual approach can be useful for figuring out the default direction you should use for an edge label. If you only use `indexFor(…).analyze()`, you could end up with a bunch of indexes that may not be needed if some of the edges are simply turned around.

The next recommendation is for when you are first setting up your production database.

**Rule of Thumb #9**

Load your data; then apply your indexes.

We recommend loading data before applying indexes because this will significantly speed up your data loading process. The application of this recommendation depends on your team's deployment strategy.

This is a common loading strategy because of the popularity of blue-green deployment patterns for production graph databases. If this is the type of pattern you would like to use, we recommend loading data and then applying indexes. For a resource on deployment strategies to minimize system downtime, like the blue-green pattern, we recommend *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* by Jez Humble and David Farley (Addison-Wesley).

There is one last tip to recommend.

**Rule of Thumb #10**

Keep only the edges and indexes that you need for your production queries.

Between development and production, you may find edge labels that you do not need for your traversals. That is expected. When you move your schema into production, get rid of the edge labels you are not going to use. Save some space on disk and the time spent persisting it.

Let's apply the new data modeling recommendations we just covered to the development schema we built up in Chapter 4. This will be the last time we use this example and sample data before we move into different graph models in future chapters.

# Production Implementation Details

The remaining implementation details in this section represent the final production version of our C360 example.

First, we will add the required materialized views to the schema for our C360 example. Then we will go through an introduction of how to load data with DataStax Bulk Loader. Last, we will revisit and update our Gremlin queries to use the new optimizations.

## Materialized Views and Adding Time onto Edges

We have a few changes to make to our development schema. First, we want to find areas where adding time onto our edges will reduce the amount of data we need to process in a query.

Let's visualize this in Figure 5-16 for the second query of our example. Figure 5-16 steps through the Gremlin query.

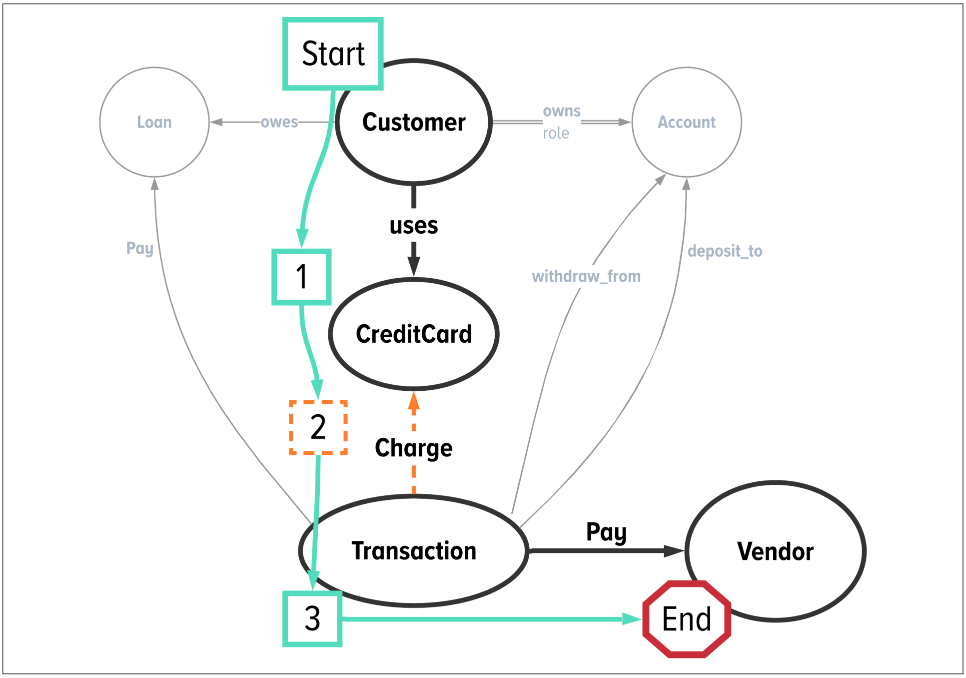



*Figure 5-16. Mapping Query 2 onto our development schema to see where we can use denormalization to minimize the amount of data we need to process*

*Example 5-5.*

```
dev.V().has("Customer", "customer_id", "customer_0").  // Start
        out("uses").                                   // 1
        in("charge").                                  // 2
        has("timestamp",                               // 2
            between("2020-12-01T00:00:00Z",            // 2
            "2021-01-01T00:00:00Z")).                  // 2
        out("Pay").                                    // 3
        groupCount().                                  // End
        by("vendor_name").                             // End
        order(local).                                  // End
        by(values, decr)                               // End
```

Comparing Figure 5-16 with Example 5-5 illustrates two production schema strategies. First, we can apply denormalization to optimize this query. Currently, time is stored only on the `Transaction` vertex. We can reduce the number of edges required in this traversal if we denormalize the `timestamp` property and store it on the `charge` edge. This is illustrated in Figure 5-16 and Example 5-5 with the label 2.

We also see in Figure 5-16 that our query walks against the direction of the `charge` edge. This means we need another materialized view on this edge label. The schema code is:

```
schema.edgeLabel("charge").
       from("Transaction").
       to("CreditCard").
       materializedView("Transaction_charge_CreditCard_inv").
       ifNotExists().
       inverse().
       create()
```

Following this same style of mapping, we can find three edge labels where denormalization can optimize our queries. This optimization minimizes the amount of data a traversal has to process by sorting the edges on disk. Specifically, we can minimize the amount of data required to process our traversals if we also add the `timestamp` property to the `withdraw_from`, `deposit_to`, and `charge` edge labels.

## Our Final C360 Production Schema

We have been exploring through schema, queries, and data integration to iteratively introduce and build up our C360 example. Together, the technical concepts and previous discussions bring us to the final production schema for our C360 example shown in Figure 5-17.

The adjustment we applied here is to denormalize and add `timestamp` onto the edge labels that we use in our traversals.

The final version of the schema code for our edge labels is shown in Example 5-6.

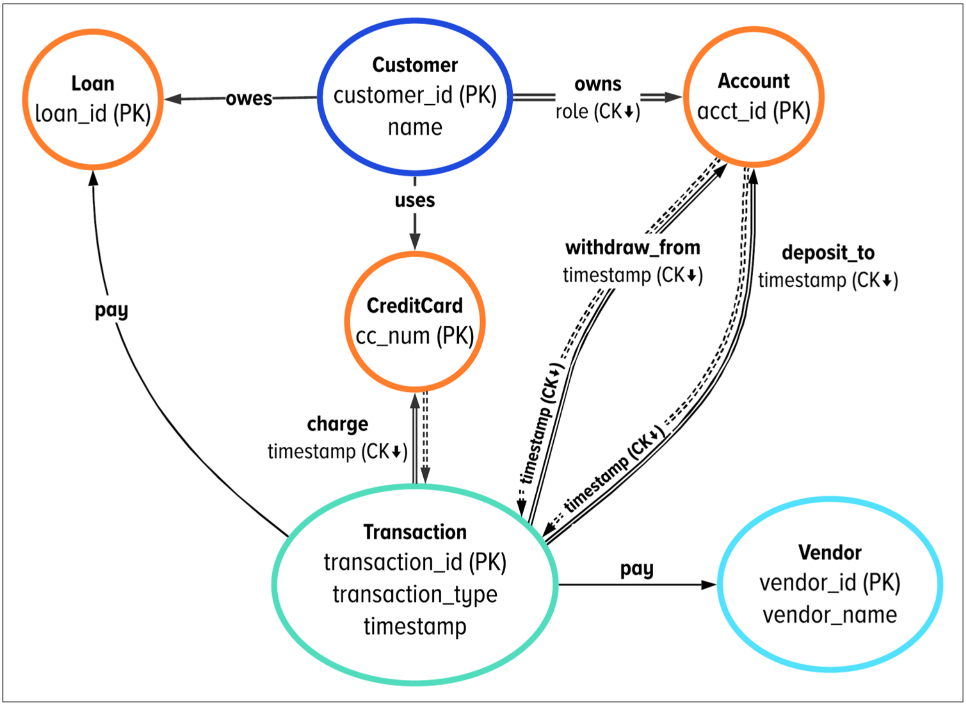



*Figure 5-17. The starting data model for a graph-based implementation of a C360 application from the previous chapter*

*Example 5-6.*

```
schema.edgeLabel("withdraw_from").
       ifNotExists().
       from("Transaction").
       to("Account").
       clusterBy("timestamp", Text). // sort the edges by time
       create();

schema.edgeLabel("deposit_to").
       ifNotExists().
       from("Transaction").
       to("Account").
       clusterBy("timestamp", Text). // sort the edges by time
       create();

schema.edgeLabel("charge").
       ifNotExists().
       from("Transaction").
       to("CreditCard").
       clusterBy("timestamp", Text). // sort the edges by time
       create();
```

To make the examples easier to follow in this book, we use `Text` to represent time and then query with strings such as `2020-12-01T00:00:00Z`. The `timestamp` property type uses less space on disk than `Text` and may be the best option for your final application.

Altogether, we need only the following changes from our development schema to our production schema:

1. Denormalize a property onto five edge labels
2. Add three materialized views to walk three edges in reverse

Let's detail how to use a bulk loading tool to insert the data into your graph database.

## Bulk Loading Graph Data

We created a script that loads all of the data into DataStax Graph from CSV files. DataStax Bulk Loader is the fastest way to load data in production. We provided a CSV file for each vertex and edge label from our data model. Let's walk through the general process for loading vertices and then show the same for edges.

### Loading vertex data with DataStax Bulk Loader

Let's look at all of the included vertex datafiles and a brief description for each file in Table 5-4.

*Table 5-4. The full list of CSV files for the vertex data used in this chapter's examples*

| Vertex file | Description |
|---|---|
| Accounts.csv | The account IDs, one per line |
| CreditCards.csv | The credit card IDs, one per line |
| Customers.csv | Customer details, one per line |
| Loans.csv | The loan IDs, one per line |
| Transactions.csv | Transaction details, one per line |
| Vendors.csv | Vendor details, one per line |

Let's see an example of how to load vertex data with DataStax Bulk Loader by examining `Transactions.csv`. The first five lines of `Transactions.csv` are shown in Table 5-5. Each line contains three pieces of information about the transaction that map to our expected schema. You also see in Table 5-5 that all transactions are loaded with an `unknown` type because one of our traversals is to mutate this property according to the graph's structure.

*Table 5-5. The first five lines of data from the file `Transactions.csv`*

| transaction_id | timestamp | transaction_type |
|---|---|---|
| 219 | 2020-11-10T01:00:00Z | unknown |
| 23 | 2020-12-02T01:00:00Z | unknown |
| 114 | 2019-06-16T01:00:00Z | unknown |
| 53 | 2020-06-05T01:00:00Z | unknown |

The most important line in Table 5-5 is the header. In the accompanying loading scripts, the header doubles as the mapping configuration between the file and the database. The header and the property names in DataStax Graph must match.

We can load the CSV file using the command-line bulk loading utility, as shown in Example 5-7.

*Example 5-7.*

```
1  dsbulk load -url /path/to/Transactions.csv
2              -g neighborhoods_prod
3              -v Transaction
4              -header true
```

Example 5-7 shows the most basic way to load vertex data on your localhost. The first part of line 1, `dsbulk load`, invokes the loading tool from the command line. The next four parameters, which can come in any order, are `-url`, `-g`, `-v`, and `-header`:

1. The `-url` parameter indicates where the CSV is stored.
2. `-g` is the name of the graph.
3. `-v` is the vertex label.
4. `-header` specifies that the data should be mapped according to the file's header.

The DataStax dsbulk documentation contains all the details for other loading options, including loading into a distributed cluster, configuration files, and much more.

Next, let's take a look at the edge data and loading process.

### Loading edge data with DataStax Bulk Loader

All of the included edge datafiles and a brief description for each are listed in Table 5-6.

*Table 5-6. The full list of CSV files for the edge data used in this chapter's examples*

| Edge file | Description |
|---|---|
| charge.csv | The `charge` edges from a `Transaction` to a `CreditCard` |
| deposit_to.csv | The `deposit_to` edges from a `Transaction` to an `Account` |
| owes.csv | The `owes` edges from a `Customer` to a `Loan` |
| owns.csv | The `owns` edges from a `Customer` to an `Account` |
| pay_loan.csv | The `pay` edges from a `Transaction` to a `Loan` |
| pay_vendor.csv | The `pay` edges from a `Transaction` to a `Vendor` |
| uses.csv | The `uses` edges from a `Customer` to a `CreditCard` |
| withdraw_from.csv | The `withdraw_from` edges from a `Transaction` to an `Account` |

Let's see an example of how to load edge data with DataStax Bulk Loader by examining `deposit_to.csv`. The first five lines of `deposit_to.csv` are shown in Table 5-7. Each line contains three pieces of information about the deposit that map to our schema: the `transaction_id`, the `acct_id`, and a `timestamp`.

*Table 5-7. The first five lines of data from the file `deposit_to.csv`*

| Transaction_transaction_id | Account_acct_id | timestamp |
|---|---|---|
| 185 | acct_5 | 2020-01-19T01:00:00Z |
| 251 | acct_5 | 2020-07-25T01:00:00Z |
| 247 | acct_5 | 2020-03-06T01:00:00Z |
| 214 | acct_14 | 2020-06-11T01:00:00Z |

The most important line in Table 5-7 is the header; the header has to match the table schema in DataStax Graph. DataStax Graph autogenerates different column names for the edge properties that are part of the table's primary key. The generated name appends the vertex label to the front of the property name, such as `Transaction_` in front of `transaction_id` and `Account_` in front of `acct_id`.

We can load the edge CSV file using the command-line bulk loading utility, as shown in Example 5-8.

*Example 5-8.*

```
1 dsbulk load -url /path/to/Transactions.csv
2             -g neighborhoods_prod
3             -e deposit_to
4             -from Transaction
5             -to Account
6             -header true
```

Example 5-8 shows the most basic way to load edge data on your localhost. The first part of line 1, `dsbulk load`, invokes the loading tool from the command line, as we saw in the previous example. The next six parameters can come in any order: `-url`, `-g`, `-e`, `-to`, `-from`, and `-header`. The `-url` parameter indicates where the CSV is stored, `-g` is the name of the graph, `-e` is the edge label, `-from` is the outgoing vertex label, `-to` is the incoming vertex label, and `-header` says to map the data according to the file's header.

The accompanying scripts show how to load all vertex and edge labels for this chapter and all examples in this book. Please head to the data directory within book's GitHub repository for the data and loading scripts for each chapter.

You will see many more examples of bulk loading data into DataStax Graph throughout the rest of the book. For now, let's move on to the next stage of our implementation details: querying our graph with Gremlin.

## Updating Our Gremlin Queries to Use Time on Edges

Now that we have updated our edge labels and indexes, let's revisit the queries and the results for each query. These are the same queries we walked through in Chapter 4, but there are two changes. First, we now can use the production traversal source `g`. We have moved out of development mode into writing queries against a production application. Second, we are going to update each query to use our new production schema. We will be using time on edges in addition to the materialized view.

Let's start by revisiting Query 1.

### Query 1: What are the most recent 20 transactions involving Michael's account?

All of the work we did to set up the schema and graph data empowers the simplicity of the query in Example 5-9 to answer our first question.

*Example 5-9.*

```
g.V().has("Customer", "customer_id", "customer_0").
      out("owns").
      inE("withdraw_from", "deposit_to"). // uses materialized view on deposit_to
      order().                            // sort the edges
        by("timestamp",desc).             // by time
      limit(20).                          // walk through the 20 most recent edges
      outV().                             // walk to the transaction vertices
      values("transaction_id")            // get the transaction_ids
```

The results remain the same, but the query processed less data by sorting the edges:

```
"184", "244", "268", ...
```

The main change from the query in Chapter 4 to this example can be seen with the addition of a single character: `E`. The query changed from using `in()` to `inE()`. This one character change uses a materialized view and the sorted order of edges.

To dig into the details, let's recall how we walked through this data in development mode. In Chapter 4, the `in()` step walked directly through edges, to the vertices, ignoring the edges' direction, and then sorted the vertex objects. That was simple enough for figuring out how to walk through our graph data.

In a production environment, we would need to ensure that this query processes only the data it needs. In Example 5-9, we optimized this query by using `inE()`, sorting all edges by time, and traversing only the 20 most recent edges.

The sorting of all edges requires three concepts from our schema. First, we use the materialized views we built on the `deposit_to` and `withdraw_from` edge labels. Second, we use the clustering key for `deposit_to` because the edges are ordered on disk by time. And last, we use the clustering key for the `withdraw_from` edge label because these edges are also ordered on disk by time.

That is a significant amount of optimization from just a small change: from `in()` to `inE()`. Let's look at what we need to do to our next query to take advantage of our new schema.

### Query 2: In December, at which vendors did Michael shop, and with what frequency?

We are going to apply the same pattern to optimize our next query. We want to take advantage of the denormalization of time on the `charge` edge to minimize the amount of data we need to process. In Gremlin, this looks like Example 5-10.

*Example 5-10.*

```
g.V().has("Customer", "customer_id", "customer_0").
      out("uses").
      inE("charge").                             // access edges
          has("timestamp",                       // sort edges
              between("2020-12-01T00:00:00Z",    // beginning of December 2020
              "2021-01-01T00:00:00Z")).          // end of December 2020
      outV().                                    // traverse to transactions
      out("pay").hasLabel("Vendor").             // traverse to vendors
      groupCount().
        by("vendor_name").
      order(local).
        by(values, desc)
```

The results are the same as before:

```
{
  "Target": "3",
  "Nike": "2",
  "Amazon": "1"
}
```

The change and optimization we applied in Example 5-10 follow the same pattern as Example 5-9. This time, we used `inE()` to access only incoming edges. We used the clustering key `timestamp` to apply a range function to the edges. Once we found all edges in a certain range, we moved to the transaction vertices and continued our traversal, as in Chapter 4.

This brings us to our last query from Chapter 4.

#### Query 3: Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

Let's think about the data this query is processing before we look at the final version of the query. In this query, we are starting from Aaliyah and finding all withdrawals from her accounts. There are no limits or time requests for this query; we want to find them all. This means that we will not be using any time ranges on the edges.

Further, every step along this query uses an existing outgoing edge label. Because of this, we do not need any materialized views and can walk out the existing edges to satisfy this query. Therefore, we need only to switch to our production traversal source, and this query will be ready to go—see Example 5-11.

*Example 5-11.*

```
g.V().has("Customer", "customer_id", "customer_4").    // accessing Aaliyah's vertex
      out("owns").                                     // walking to the account
      in("withdraw_from").                             // Only consider withdraws
      filter(
             out("pay").              // walking out to loans or vendors
             has("Loan", "loan_id", "loan_18")).       // only keep loan_18
      property("transaction_type",    // mutating step: set the "transaction_type"
                "mortgage_payment").  // to "mortgage_payment"
      values("transaction_id", "transaction_type")     // return the id and type
```

The results look exactly the same as those in Chapter 4:

```
"144", "mortgage_payment",
"153", "mortgage_payment",
"132", "mortgage_payment",
...
```

With Example 5-11, we have concluded the transformation from development to our production schema and queries. We encourage you to apply the thought process of shaping query results from "Advanced Gremlin: Shaping Your Query Results" on page 106 to create more robust payloads and data structures to share within your application.

# Moving On to More Complex, Distributed Graph Problems

We consider the transition from Chapter 4 to the topics and production optimizations presented in this chapter to be the final stage of learning how to work with graph data in Apache Cassandra. Along the way, you experienced limitations, followed by their resolutions. We will see more of that as we go along but in shorter iterations.

## Our First 10 Tips to Get from Development to Production

Throughout Chapter 4, we presented data modeling tips for mapping your data into a distributed graph database. In this chapter, we augmented those tips with specific ways to optimize your production graph database. Let's revisit all 10 tips to recall the journey we went through from development to production (Figure 5-18).

These 10 tips are foundational to starting over with a new dataset and use case. We will be applying them repeatedly in the coming chapters. And we will find more recommendations to add to this list as we explore different common structures for distributed graph applications.

| No. | Tip |
|---|---|
| 1 | If you want to start your traversal on some piece of data, make that data a vertex. |
| 2 | If you need the data to connect concepts, make that data an edge. |
| 3 | Vertex-Edge-Vertex should read like a sentence or phrase from your queries. |
| 4 | Nouns and concepts should be vertex labels. Verbs should be edge labels. |
| 5 | When in development, let the direction of your edges reflect how you would think about the data in your domain. |
| 6 | If you need to use data to sub-select a group, make it a property. |
| 7 | Properties can be duplicated onto edges or vertices use denormalization to reduce the number of elements you have to process in a query. |
| 8 | Let the direction you want to walk through your edge labels determine the indexes you need on an edge label in your graph schema. |
| 9 | Load your data; then apply your indexes. |
| 10 | Only keep the edges and indexes that you need for production queries. |

*Figure 5-18. Our top 10 graph data modeling tips*

From here, we think you are ready to tackle deeper and more complex graph problems such as paths, recursive walks, collaborative filtering, and more.

The most advanced graph users today are those who are willing to learn through trial and error. We have collected what they have learned so far and will be walking you through those details within the context of new use cases in the coming chapters.

As we see it, gaining traction with new technology and new ways of thinking is a journey. We have presented the major foundational milestones others have reached so far. Now, you are ready to come along with us and apply graph thinking in production applications to solve complex problems.

In Chapter 6, we'll look at one of the most popular ways for people to extend graph thinking into their data. We will solve a complex problem found at the intersection of edge computing and hierarchical graph data in a self-organizing communication network of sensors.

# CHAPTER 6
# Using Trees in Development

C360 applications for neighborhood exploration are the most popular use of distributed graph technology at this time. A C360 example also serves as a great introduction to a plethora of concepts in distributed systems, graph theory, and functional query languages.

But what else is out there?

In the next two chapters, we step beyond understanding neighborhoods of data and apply graph thinking to hierarchical data.

*Hierarchical data*
: Hierarchical data represents concepts that naturally organize into a nested structure of dependencies.

At the time of writing this chapter, hierarchically structured data is the second most popular shape of data used in distributed graph applications.

## Chapter Preview: Navigating Trees, Hierarchical Data, and Cycles

There are five main sections to this chapter.

The first section walks through multiple examples of hierarchical data from real-world scenarios. With a new shape of data comes another flood of terminology; the second section introduces new terms with many examples. The third section of the chapter introduces the problem statement, data, and schema we will use in our examples. With our data, there are two main styles of queries for working with hierarchical data. The fourth section explains the first query pattern: walking from the bottom of

the hierarchy to the top. The last section shows the second query pattern: walking from the top of the hierarchy to the bottom.

The final query pattern in the last section unveils one of the most difficult aspects of working with deeply nested data in a production application. We end this chapter showing how things can break, setting the stage for Chapter 7, in which we explain why and how to fix them for production.

## Seeing Hierarchies and Nested Data: Three Examples

More often than not, we already use graphs to describe the natural, nested structure within concepts we use every day. We often see hierarchical structure within the data about a product's structure, version control systems, or people. Let's dive into each of these three examples and illustrate how we reason about nested data with a graph.

### Hierarchical Data in a Bill of Materials

The first place to explore natural hierarchies within data can be seen in any bill of materials (BOM) application. A BOM application describes a product's structure by associating the nested dependences of the raw materials, assemblies, parts, and quantities needed to create a product in an end-to-end pipeline. Figure 6-1 illustrates the dependencies for constructing a Boeing 737 airplane.

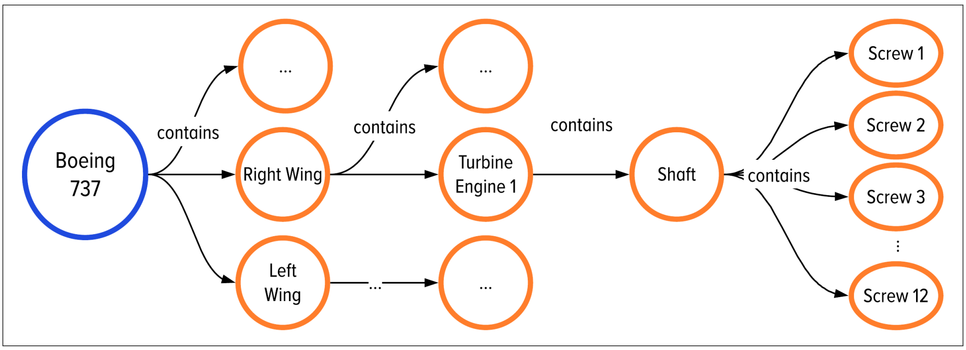



*Figure 6-1. An example of hierarchical data in a bill of materials example*

You can see the natural hierarchy or "nestedness" of data when you consider the BOM required to build an airplane. Consider this question: how many screws are used to construct a Boeing 737? The answer can be found by walking through the hierarchy of components that are assembled to construct a plane: a plane has two wings, each wing has one turbine engine, the engine has a shaft that requires 12 screws, and so on.

When we talk about hierarchies in a BOM, we are talking about following that same deconstruction for every part of the plane to figure out the total number of screws it

takes to build the whole object. This type of hierarchy in data exists for manufacturing plants, assembly lines, and myriad areas within industrial engineering.

## Hierarchical Data in Version Control Systems

You also find hierarchies and graph data structures in software engineering processes. The most popular one, and the one used to supplement this book with technical content, is Git.

Git's version control system forms a hierarchy. You can think of this version control system as containing three separate tree structures: the working directory, the index, and the head. Each tree in the version control system has a different and specific purpose: writing, staging, or committing changes. To illustrate this, Figure 6-2 shows how a dependency graph for your project is observable between each state of changes.

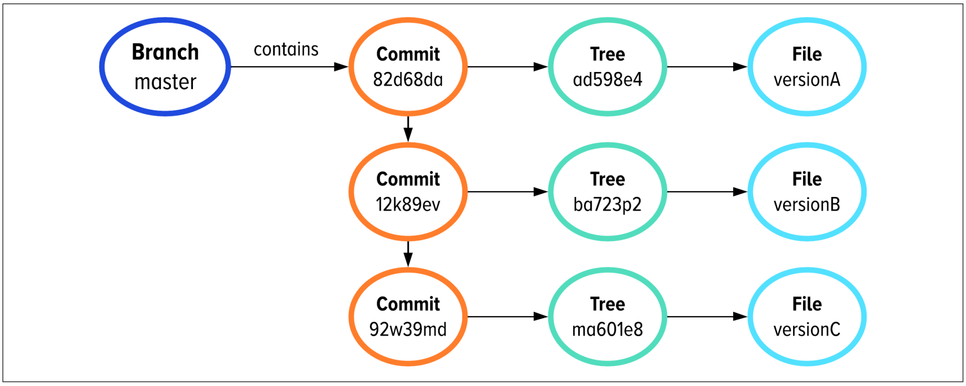



*Figure 6-2. An example of hierarchical data in version control within software development*

You can also think of Git as a chain. In this light, the version control system creates a chain of dependencies with forks. Either way you prefer to think of it, the shape of data within Git's version control system forms a nested hierarchy.

Let's look at a third example where we find hierarchical structure in data.

## Hierarchical Data in Self-Organizing Networks

The last example of natural hierarchies can be found in how people self-organize. There are two main examples of this: family trees and corporate hierarchies. To really bring home hierarchies and their relationships to graph data structures, think about your own family. Think back as far as you can, maybe to a great-great-grandparent. Tracing your family's lineage from a long-ago ancestor to you forms a hierarchy of parents and children across many levels. The parent-child dependency within a family is one of the best examples of hierarchy in natural data.

We create the same type of organization within our workforces; an example corporate hierarchy is shown in Figure 6-3.

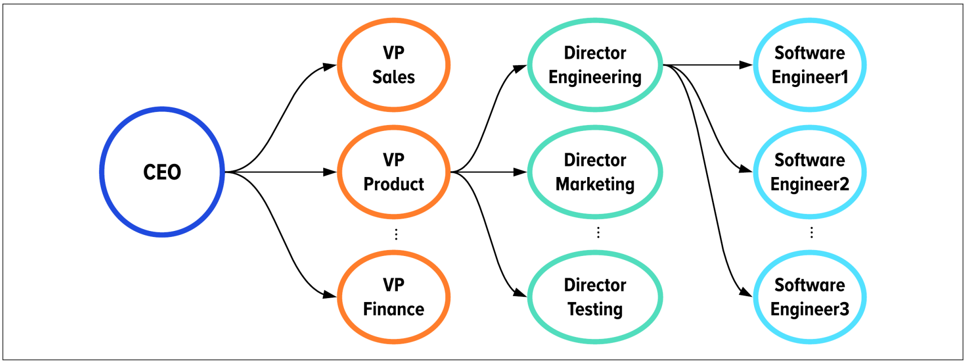



*Figure 6-3. An example of hierarchical data in a corporate structure*

Corporate structures look somewhat similar to family trees. The manager-employee relationship is the same as your family's parent-child relationship. We work in groups and organize ourselves in the same structure as our lineage. Broadly speaking, a CEO has a team of vice presidents, each vice president has a team of directors, and directors manage teams of individual contributors.

It is great to realize how we already use nested relationships to describe common concepts, but let's explore why this shape of data is currently the second most popular use of graph technology.

## Why Graph Technology for Hierarchical Data?

Graph technology enables a more natural way to represent the nested relationships within data. The more natural representation of data yields simpler code to maintain and makes development teams more productive.

For example, during one of the many conversations we had with graph users around the world for this book, we found an early adopter who told us that his team "translated 150 lines of a query on top of HBase into 20 lines of Gremlin." This is exactly why engineering teams are adopting graph technology to model, store, and query hierarchically structured data.

The simplification to the codebase, and the resulting enhancement to developer productivity, has been a common theme in our conversations with users. This is encouraging more teams to use distributed graph technology to model, reason, and solve complex problems with natural hierarchies.

So what do corporate structures, version control, and product structures have in common?

When we look at the data for each of these concepts, we see nested or hierarchical data. When using graph technologies, these hierarchies are called trees.

To lay the foundation for what we see, let's take a tour of a new wave of graph terms so that we can teach you how to see the trees within this forest of data.

# Finding Your Way Through a Forest of Terminology

The definitions throughout this section bring together terminology from the database and graph-theoretic communities. Concepts about the data's storage model, like hierarchy, are popular terms about databases. Terms that define observable structures within the data, such as *tree* and *forest*, originate in graph theory.

Where the terms come from does not matter. Being able to distinguish between concepts related to storage versus those related to sample data does matter. We already ran into how easy it can be to confuse concepts from graph data and graph schema in Chapter 2. We see the same confusion again with hierarchical data. The constant mixture of terminology from multiple communities explains why graph technology can be difficult to pick up.

To help you navigate both worlds, let's look at some examples that can put a picture to some key terms.

## Trees, Roots, and Leaves

We have used the term *tree* a few times without defining it. Let's do that now.

*Tree*
: A tree is a connected graph with no cycles.

We will formally define a cycle in the next section. For now, let's revisit our example corporate hierarchy to see trees in practice. The graph in Figure 6-4, from the CEO down to the software engineers, forms a tree.

Examining the edges in Figure 6-4 shows that every vertex has only one edge pointing to it. If you modeled your company's corporate tree and compared its structure to your competitor's corporate tree, you would be looking at two separate trees. Those two trees together make a forest. Yes, mathematicians had a bit of fun when coming up with these official graph theory terms; let the puns begin.

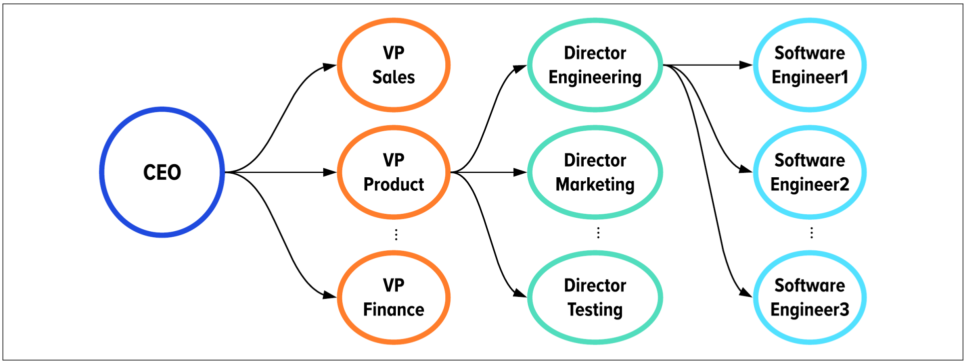



*Figure 6-4. A visualization of a corporate hierarchy as an example of a tree from graph theory*

There are two special types of vertices within hierarchical data: parents and children.

*Parent vertex*
: A parent vertex is one step higher in the hierarchy.

*Child vertex*
: A child vertex is one step below a parent in the hierarchy.

You can identify examples of these terms in Figure 6-4. The VP of Product in Figure 6-4 is the parent of the Director of Marketing. The Director of Marketing is a child vertex of the VP of Product.

The following definitions explain how roots and leaves fit into the traditional understanding of parent and child dependencies in hierarchical data.

*Root*
: A root is the topmost parent vertex; a root is the beginning of the dependency chain within a hierarchy.

*Leaf*
: A leaf is the last child vertex in a dependency chain within a hierarchy; a leaf vertex has a degree of one.

Looking at the diagram in Figure 6-4, the CEO is the root, and each software engineer is a leaf.

## Depth in Walks, Paths, and Cycles

Data within a hierarchy is usually referenced in one of three ways in an application: by its neighborhoods, by its depth, or by its path.

First, an application references hierarchical data according to its parents or its children. From a certain vertex, you would walk up one level to report the parent vertex,

or you would walk down one level to report its children. This is very similar to walking around neighborhoods like we have been doing in the past few chapters.

Second, an application references hierarchical data according to its distance from either a root or a leaf. We use the term *depth* to refer to this distance in hierarchical data.

*Depth*
: In a hierarchy, depth is the distance of any vertex in the graph to its root; the maximum depth in a tree is found from its root.

Let's take a look at our corporate hierarchy tree to apply depth to this data.

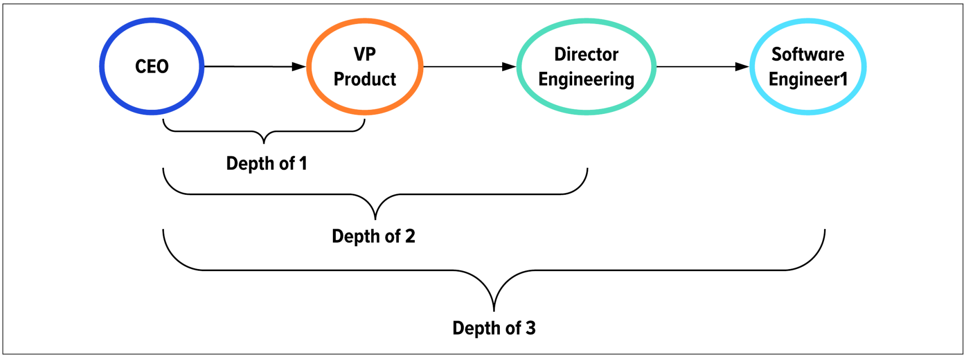



*Figure 6-5. Using a corporate hierarchy to understand depth in tree data*

While you have been thinking about corporate reporting structures, you probably have been considering how far each position is from the CEO. Figure 6-5 gives us a formal terminology for that natural association. Looking at the hierarchy in Figure 6-5, we say that the VP of Product is 1 away from the CEO. The Director of Engineering has a depth of 2 from the CEO. Last, a software engineer has a depth of 3 from the CEO.

The third way that ,.hierarchical data is used in an application requires understanding the full dependency chain between two pieces of data. Accessing the full dependency chain requires traversing through the data from the root to the leaves or vice versa. This brings us to three useful terms.

*Walk*
: A walk through a graph is a sequence of visited vertices and edges. Vertices and edges *can* be repeated.

*Path*
: A path through a graph is a sequence of visited vertices and edges. Vertices and edges *cannot* be repeated.

*Cycle*
A cycle is a path where the starting and ending vertices are the same.

Let's look at Figure 6-6, which shows an example of a path from the root to a leaf in our corporate tree.

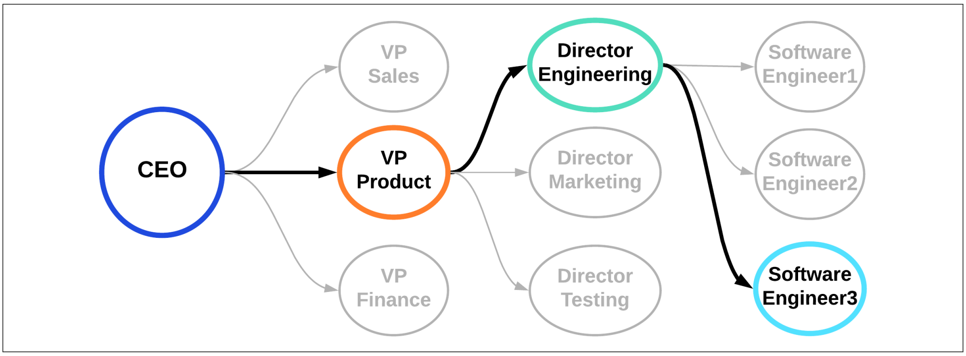



*Figure 6-6. Walking through a corporate hierarchy to show a path from its root, the CEO, to a leaf, a software engineer*

The path in Figure 6-6 walks from the CEO through two different levels to get to a software engineer. This is a path because all data along the way is used only once. In other words, there are no repeated edges or vertices in this example path: CEO → VP of Product → Director of Engineering → Software Engineer 3.

The natural translation of hierarchical data into how we think and reason about it is exactly why teams are using graph technology. The way that we represent, store, and query hierarchical data with graph technology already follows how we think about it, naturally!

Now that we understand the terminology, let's set up the example we will be using in the next two chapters.

## Understanding Hierarchies with Our Sensor Data

If you use electricity, you likely contribute to a distributed hierarchy of data every moment of your day.

On an hourly basis, you contribute to distributed, hierarchical graph data structures by flicking a light switch in your house or business. Your power supplier tracks how much energy your home or workplace uses on a time interval, likely every 15 minutes. These readings are collected and sent back to your power company, which aggregates them.

Your power company may even distribute these readings from one power recipient to another via a self-organizing network of sensors within the power chain. The transfer

of these readings through a self-organizing network is one of the most beautiful, dynamic, and hierarchical graph problems we interact with on a constant basis.

The example in this chapter models the dynamic and hierarchical network of communication found within a self-organizing network of sensors and towers, much as how voltage levels are communicated from your home to your power company.

To bring this example to life, we are asking you to think like a data engineer for a fictitious power company, Edge Energy. Your objective will be to understand, model, and query the hierarchical structure found within Edge Energy's communication network.

We advise teams to approach any new problem like this one in three steps:

1. Understand the data.
2. Build a conceptual model using the GSL notation.
3. Create the database schema.

The next three sections follow these steps.

## Understand the Data

Each reading collected by Edge Energy at any home or business is reported for a few different compliance scenarios, like real-time auditing. One of the most complex problems the company has to prepare for is: what if one of the communication towers goes down?

To help you envision this, consider the zoomed-in snapshot of Edge Energy's network in Figure 6-7.

Figure 6-7 shows Edge Energy's sensors (the asterisks) and communication towers (the diamonds); we have highlighted one of the towers in orange. Ultimately, our example across the next two chapters has to answer this question: what would happen to Edge Energy's sensor data if the orange tower went down? That is, Edge Energy wants to assess the impact of a tower's failure on the accessibility of sensor data across the entire network so that the company can prepare for different failure scenarios.

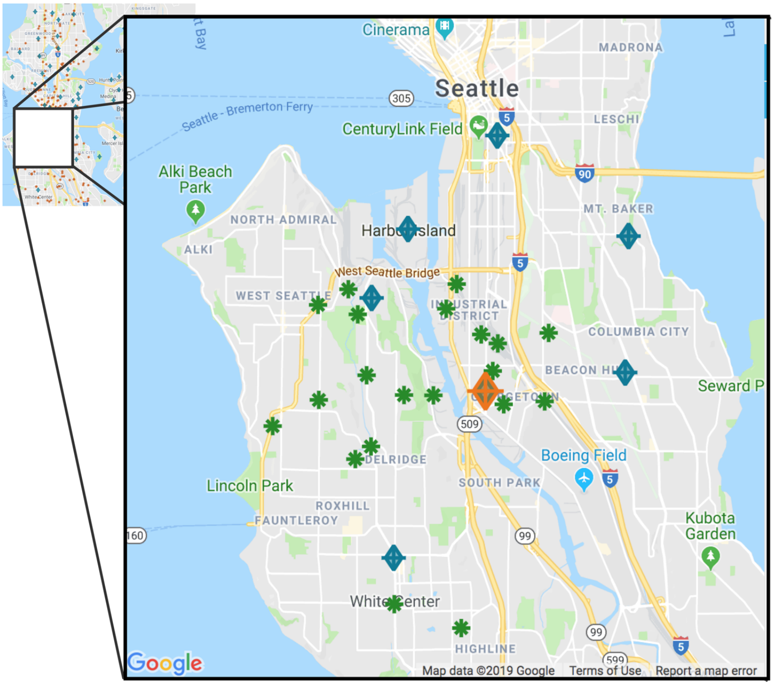



*Figure 6-7. Visualizing the sensors and towers used by Edge Energy around Georgetown, a neighborhood in Seattle, WA (etwork edges are not shown for image clarity)*

The problem requires that we first understand a single tower. If we can understand one tower, we can understand any of the towers on the network. And the answer we get to at the end of Chapter 7 may surprise you.

Let's walk through how a dynamic and hierarchical graph is constructed in Edge Energy's network of sensors and towers.

In Edge Energy's network, the sensors are responsible for two things. First, a sensor takes readings of the residence or business to which it is assigned. Second, on a time interval, every sensor communicates its reading to another available point in the network—either a nearby sensor or a tower. The objective is for every reading to eventually pass through this network to a tower and back to Edge Energy's monitoring system.

In Figure 6-8 we have zoomed in to look at the network in a different area of Seattle.

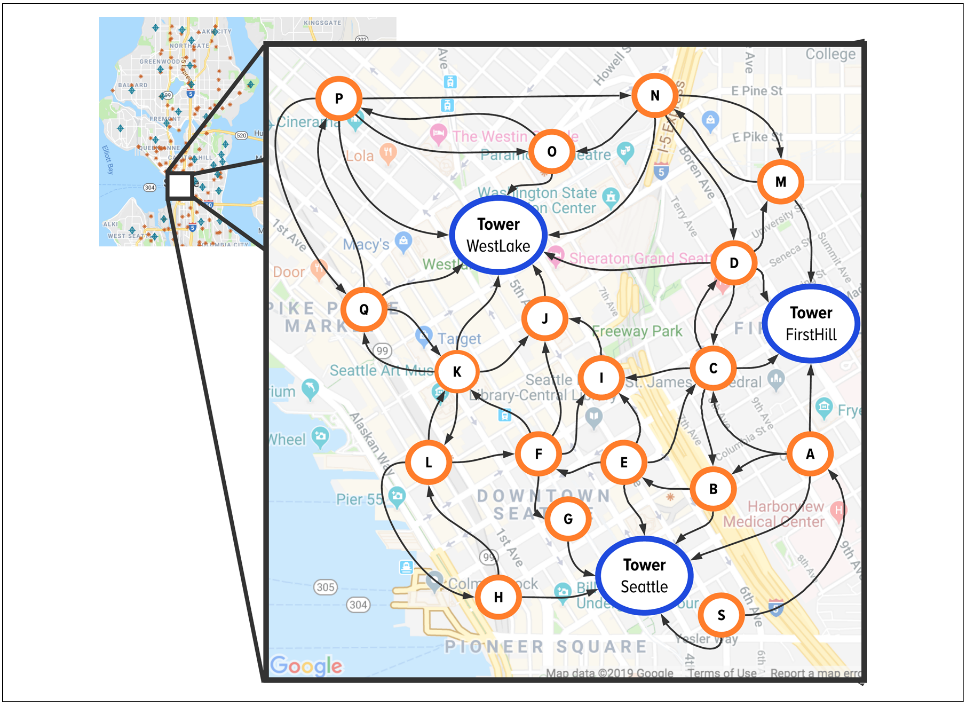



*Figure 6-8. Zooming in to show the communication network within the downtown Seattle area*

What you won't see in Figure 6-8 is the hierarchical nature of the data, but you will see it in how we use the data (coming up).

As we recently talked about, applications that use hierarchical data query the data in two main patterns: from the bottom up or from the top down. It is in how the communication data is used that its hierarchical structure becomes easier to see.

### Seeing hierarchies in data: From the bottom up

We are going to spend time walking through and understanding our data before we write code to query it.

The first way we want to use the data in Edge Energy's sensor network is to understand how the data from a sensor reached a tower. Let's take a look in Figure 6-9 at how the data from `Sensor S` was shared throughout the network to pass its reading to a tower.

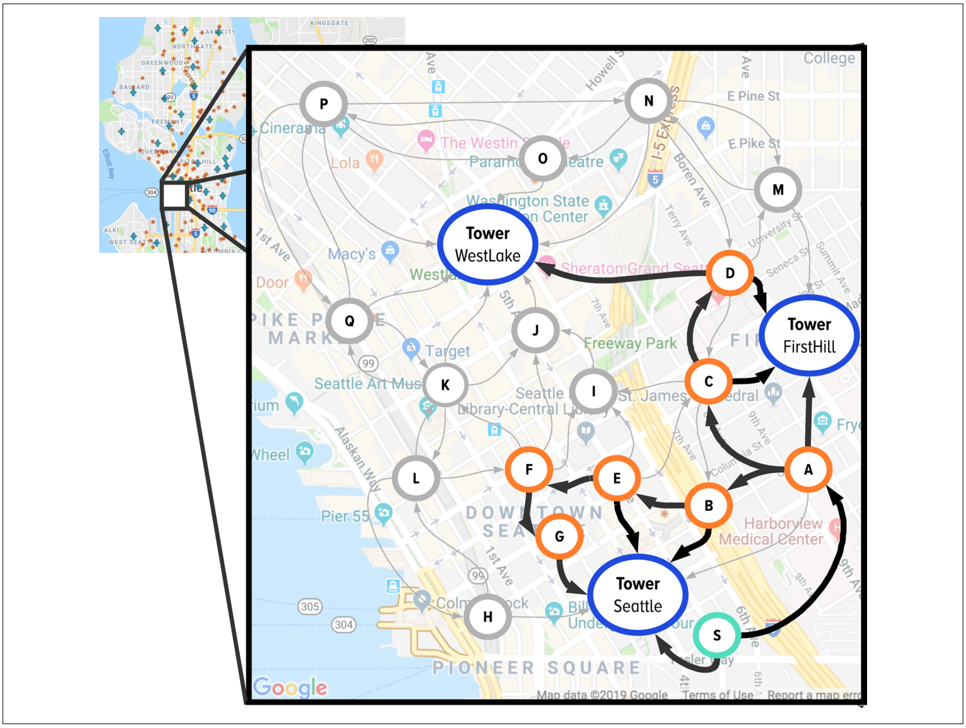



*Figure 6-9. Looking at our example data to walk from a sensor up to a tower*

Figure 6-9 emphasizes one traversal: from `Sensor S` to nearby towers over the course of an entire day. If you trace through every walk, you will find many unique ways to walk from `Sensor S` to any tower. Example paths include:

```
S → Seattle
S → A → FirstHill
S → A → C → FirstHill
S → A → C → D → FirstHill
S → A → C → D → WestLake
```

To look at this in a different way, Figure 6-10 shows the hierarchical structures from Figure 6-9.

Looking at the data in Figure 6-10 illustrates the unbounded and hierarchical nature of the data. Some paths from `Sensor S` have a distance of 1 whereas others vary up to a distance of 5. Figure 6-11 shows how you can quickly find the distance of a path in this hierarchy.

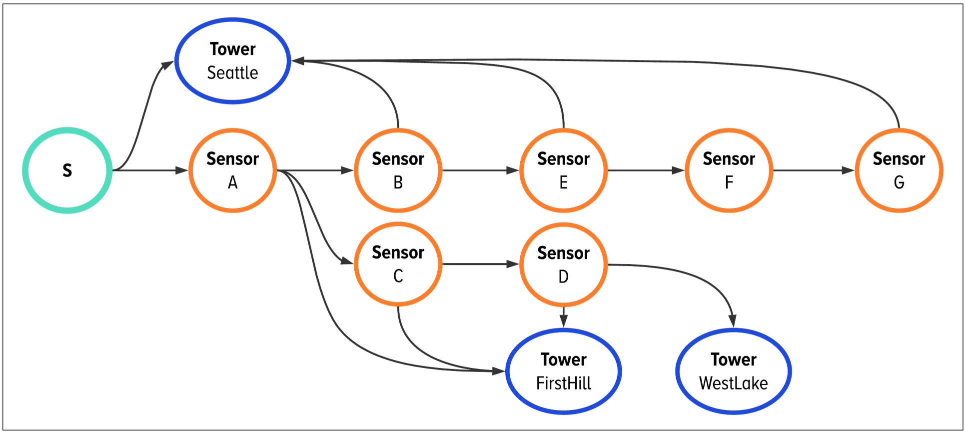



*Figure 6-10. Showing the hierarchy from `Sensor S` multiple tower vertices*

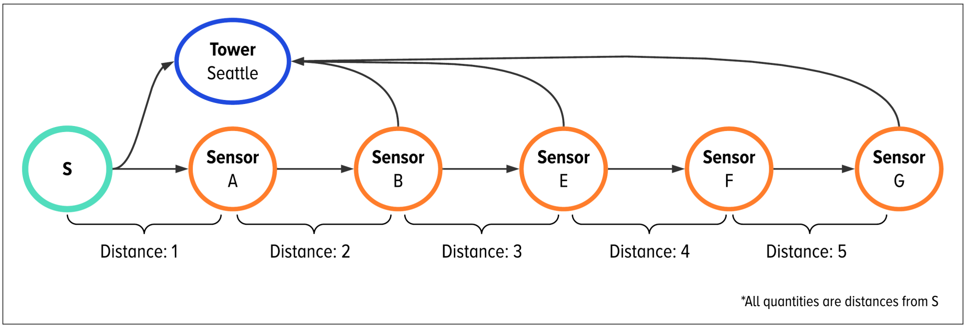



*Figure 6-11. Understanding a path's distance in our example data.*

Figure 6-11 shows that the distance from `Sensor S` to the `Seattle` tower can be 1, 3, 4, or 6. The path of length 1 is `Sensor S → Seattle`. The path of length 3 is `Sensor S → A → B → Seattle`. The path of length 4 is `Sensor S → A → B → E → Seattle`. The path of length 6 is `Sensor S → A → B → E → F → G → Seattle`.

Through some hierarchy, every sensor's reading ultimately reaches a tower.

In the real world, these sensors are free to communicate with any nearby sensor or tower. This means that the hierarchical structures within our graph are dynamic and constantly changing. These dynamic networks create some of the most beautiful mixtures of time series data with graph structures in Cassandra.

Now that we understand how to see them from the bottom up, let's reverse the direction and explore dynamic networks from the towers down to sensors.

### Seeing hierarchies in data: From the top down

The second way we will be querying this data is from the top down: from towers to sensors. Figure 6-12 zooms in on our example data to show the data reachable in two steps from the `WestLake` tower.

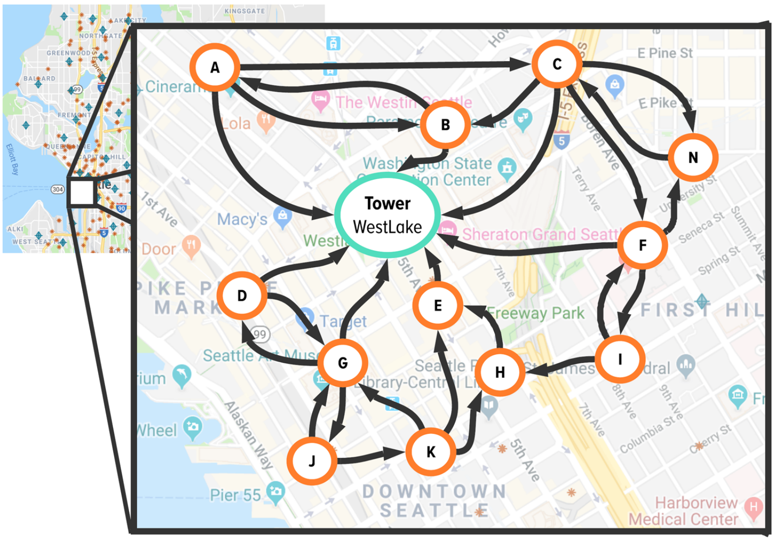



*Figure 6-12. Illustrating the second neighborhood of sensors that connect to the `West Lake` tower*

Figure 6-12 shows the sensors that are reachable in a walk of length 2 from the `West Lake` tower. Examining the edges, we see that sensors `A`, `B`, `C`, `F`, `E`, `G`, and `D` are in the first neighborhood of the `WestLake` tower. In hierarchical data, we say sensors `A, B, C, F, E, G,` and `D` have a depth of 1 from the root, `WestLake`. Sensors `J`, `K`, `H`, `I`, and `N` are in the second neighborhood of the `WestLake` tower. In hierarchical data, we say sensors `J`, `K`, `H`, `I`, and `N` have a depth of 2 from the root, `WestLake`.

The hierarchical structure, and each sensor's depth, may be easier to see in Figure 6-13.

Figure 6-12 and Figure 6-13 show the same data. We are looking at how to traverse our data from the top of the hierarchy to the bottom.

One of the most important concepts to realize is that the example here represents real-world hierarchies. They are not perfect trees. These hierarchies are messy; they contain cycles.

To see that, let's talk about how an edge is created in this dataset and in its real-world version.

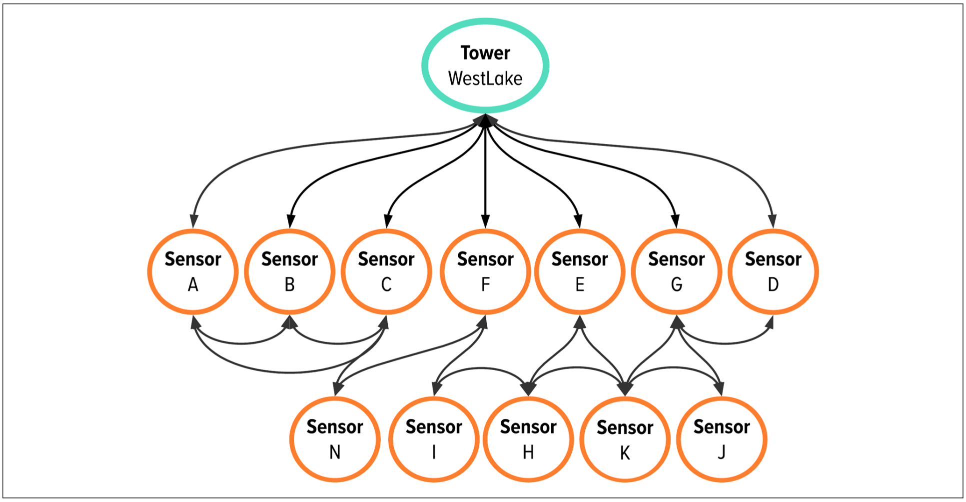



*Figure 6-13. Understanding depth from the root vertices in our example data*

### Understanding edges in the sensor hierarchies

The queries in the upcoming sections will be walking up and down the sensor communication hierarchies. The following rules apply to the presence of edges between sensors and towers:

1. Edges start from any sensor and go to a neighboring sensor or tower.
2. There can be no loops; a sensor cannot add an edge to itself.

> Loops are different from cycles. A loop is an edge that starts and ends at the same vertex; a cycle is a series of edges that start and end at the same vertex. There may be cycles in these network but never loops.

We apply the hierarchical network of edges in our dataset to show how Edge Energy uses the edges in its application:

1. Edges chain together to create walks.
2. Walks represent communication from a sensor to a tower.
3. Walks start at a sensor and end at a tower, and vice versa.

At this point, we have completed step one of our three steps. We are moving on from understanding the data to query-driven data modeling.

## Conceptual Model Using the GSL Notation

With our example and the data it provides, we aim to gain insight into the dynamic network formed by Edge Energy's sensors. We will want to report the paths used to share a sensor's reading to a tower so that we can understand failure scenarios. To do that, we will focus on addressing the following queries:

1. What path did a sensor's data follow to pass its information to a tower?
2. What sensors communicated with a specific tower?
3. What is the impact of the shutdown, loss, or general failure of a tower?

Combining our understanding of the data, the queries listed above, and data modeling recommendations from previous chapters, we arrive at a very basic database schema for our example, as shown in Figure 6-14.

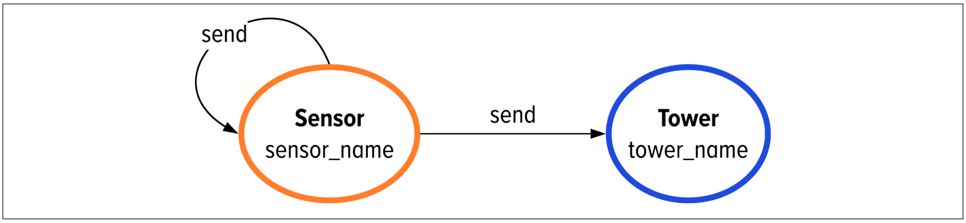



*Figure 6-14. The starting development schema for our example in this chapter*

Figure 6-14 applies query-driven modeling along with our data modeling best practices to arrive at a graph database schema. As we have done throughout this book, we created two vertex labels that represent the main entities of interest in this data: Sensors and Towers. To show how a sensor communicates with Edge Energy, we have an edge label called `send` from a Sensor vertex label to the Tower vertex label. To illustrate how sensors communicate with each other, we have a self-referencing edge label `send` that starts and ends with the Sensor vertex label.

Recall from Chapter 2 that self-referencing edge labels are different from loops. Self-referencing edge labels represent schema elements that start and end at the same vertex label. This is different from a loop, which is a concept in the data, not the schema. Loops are edges in data that start and end with the vertex—like an edge starting and ending at `Sensor 1`. We will not have loops in our data, and consequently, sensors will not be allowed to send information to themselves.

## Implement Schema

The accompanying dataset represents real towers and sensors across the broader Seattle area. For Edge Energy, this is just one small area of its global network.

Each tower in the dataset represents a real cell phone tower. Each tower has a unique identifier, a name, and a geo-location. We already saw this when we talked about the `WestLake` tower. The same is true for the sensors. The sensors have a unique identifier and a valid geo-location around the Seattle area. We have been using letters to identify a sensor in our examples, like `Sensor A`, but the identifiers in the real dataset are integers.

A new feature we have in our example is the ability to reference the geo-location of a specific vertex. We do this by creating points in the schema code:

```
schema.vertexLabel("Sensor").
       ifNotExists().
       partitionBy("sensor_name", Text).
       property("latitude", Double).
       property("longitude", Double).
       property("coordinates", Point).
       create();

schema.vertexLabel("Tower").
       ifNotExists().
       partitionBy("tower_name", Text).
       property("latitude", Double).
       property("longitude", Double).
       property("coordinates", Point).
       create();
```

There are only two edge labels that we need to create for our example. We need to model a sensor sending information to either another sensor or a tower. The schema code will be:

```
schema.edgeLabel("send").
       ifNotExists().
       from("Sensor").
       to("Sensor").
       create()

schema.edgeLabel("send").
       ifNotExists().
       from("Sensor").
       to("Tower").
       create()
```

### Loading vertex data with DataStax Bulk Loader

Let's look at all of the included vertex data files and a brief description for each in Table 6-1.

*Table 6-1. The full list of CSV files for the vertex data used in this chapter's examples*

| Vertex file | Description |
|---|---|
| Sensor.csv | The sensors, one per line |
| Tower.csv | The towers, one per line |

Let's see an example of how to load vertex data with DataStax Bulk Loader by examining `Tower.csv`. The first five lines of `Tower.csv` are shown in Table 6-2.

*Table 6-2. The first five lines of data from the file Tower.csv*

| tower_name | coordinates | latitude | longitude |
|---|---|---|---|
| Renton | POINT (-122.203199 47.47896) | 47.47895812988281 | -122.20320129394 |
| MapleLeaf | POINT (-122.322603 47.69395) | 47.69395065307617 | -122.32260131835 |
| MountainlakeTerrace | POINT (-122.306926 47.791277) | 47.79127883911133 | -122.30692291259 |
| Lynnwood | POINT (-122.308106 47.828134) | 47.82813262939453 | -122.30810546875 |

In the accompanying loading scripts, the header doubles as the mapping configuration between the file and the database. The header and the property names in DataStax Graph must match.

We can load the CSV file using the command-line bulk loading utility as shown in Example 6-1.

*Example 6-1.*

```
dsbulk load -url /path/to/Tower.csv
            -g tree_dev
            -v Tower
            -header true
```

Example 6-1 shows the most basic way to load vertex data on your localhost, just like we did in Chapter 5. Next, let's look at the edge data and loading process.

### Loading edge data with DataStax Bulk Loader

All of the included edge datafiles and a brief description for each are listed in Table 6-3.

*Table 6-3. The full list of CSV files for the edge data used in this chapter's examples*

| Edge file | Description |
|---|---|
| Sensor_send_Sensor.csv | The `send` edges between sensors in this example |
| Sensor_send_Tower.csv | The `send` edges between sensors and towers in this example |

Let's see an example of how to load edge data with DataStax Bulk Loader by examining `Sensor_send_Sensor.csv`. The first five lines of `Sensor_send_Sensor.csv` are shown in Table 6-4.

*Table 6-4. The first five lines of data from the file `Sensor_send_Sensor.csv`*

| out_sensor_name | timestep | in_sensor_name |
|---|---|---|
| 103318117 | 1 | 126951211 |
| 1064041 | 2 | 1307588 |
| 1035508 | 2 | 1307588 |
| 1282094 | 1 | 1031441 |

The most important line in Table 6-4 is the header; the header has to match the table schema in DataStax Graph. DataStax Graph autogenerates different column names for the edge properties that are part of the table's primary key. The header line in Table 6-4 shows how DataStax Graph appends `out_` and `in_` to the front of the partition key columns in the case of a self-referencing edge. If you would like to discover this on your own, you can use your schema tools inside of DataStax Studio or `cqlsh` to inspect the naming conventions of your schema.

You also see a property called `timestep` in Table 6-4, but our schema does not have this property on our edges in the database. In this case, the extra data will be ignored during the loading process; we will not end up with `timestep` on our edges even though it is in the data.

> We will revisit and use the `timestep` property in Chapter 7 when we introduce how to apply time to our data and how to use it in your traversals. To add in all of that complexity now is too much for what we want to cover at this point in the development of this example.

We can load the edge CSV file using the command-line bulk loading utility as shown in Example 6-2.

*Example 6-2.*

```
dsbulk load -url /path/to/Transactions.csv
            -g trees_dev
            -e send
            -from Sensor
            -to Sensor
            -header true
```

Example 6-2 shows the most basic way to load edge data on your localhost, as we saw in Chapter 5. The accompanying scripts show how to load all vertex and edge data for this chapter and all examples in this book. Please refer to the data directory within this book's GitHub repository for the data and loading scripts for each chapter.

## Before We Build Our Queries

So far, we have accomplished three tasks for our example in this chapter. We explored the data we will be using for this example. Then, we built a model for sensors and towers to trace communication throughout a network of sensors. Last, we loaded the data to use for our upcoming queries.

In graph applications, querying and using tree structures primarily focuses on traversing up and down the tree's structure. When we say we are traversing up the tree structure, we are talking about walking up from a leaf to the root. Traversing down the tree structure goes in the opposite direction: from the root down to a leaf or leaves.

Let's iron out the concepts and queries by walking up and down the sensor trees in development mode. We will start with showing how Edge Energy can follow a sensor's communication path to a tower by walking up the trees.

We will unveil the reason one way is harder than the other at the end of this chapter, setting the stage for Chapter 7.

Now, we are ready to write queries.

# Querying from Leaves to Roots in Development

The upcoming examples apply the data model to answer the queries for Edge Energy. Our first question queries the data from the leaves up to the root to answer the following:

- What path did a sensor's data follow to pass its information to a tower?

We are breaking this question down into two steps:

1. Where has a specific sensor sent information to?
2. What was this sensor's path to any tower?

Answering each of these questions builds up to showing how to query from leaves to roots in hierarchical data. Let's dive in and see how to do this with Gremlin.

## Where Has This Sensor Sent Information To?

This first query asks to explore the neighborhoods of data accessible from a given sensor. We picked `Sensor 1002688` for this example. We want to start with understanding the first neighborhood; Example 6-3 shows the query and Example 6-4 displays the results.

> The step `dev.V(vertex)` compiles to the same query as `dev.V().hasLabel(label).has(key, value).has(key, value)…` and so on. A `has()` clause is required for every property in the vertex's primary key.

*Example 6-3.*

```
1 sensor = dev.V().has("Sensor", "sensor_name", "1002688"). // look up the sensor
2                  next()            // return the sensor vertex
3 dev.V(sensor).                     // look up the sensor
4     out("send").                   // walk through all send edges
5     project("Label", "Name").      // for each vertex, create map with two keys
6       by(label).                         // the value for the first key "Label"
7       by(coalesce(values("tower_name",    // for the 2nd key "Name": if a tower
8                          "sensor_name"))) // else, return the sensor_name
```

*Example 6-4.*

```
{
  "Label": "Sensor",
  "Name": "1035508"
},{
  "Label": "Tower",
  "Name": "Georgetown"
}
```

Example 6-3 and Example 6-4 explore the first neighborhood for `Sensor 1002688`. Lines 1 through 3 illustrate another way to access and use vertex objects with DataStax Graph. Lines 4 through 8 query the first neighborhood and shape the result set. The results show that `1002688` sent data to one sensor and one tower: `1035508` and `Georgetown`. This means that `Sensor 1002688` is nearby and communicated with

those `Sensor 1035508` and the `Georgetown` tower throughout the entire scope of the sample data.

Line 3 in Example 6-3 introduces one new concept: direct vertex lookup with the `V(vertex)` syntax. We did this to show how to store an object in your application's memory and use it in a traversal; it might be useful for you at some point in your application's development.

If you feel comfortable with applying these steps and shaping the query results, you can skip ahead to the next query.

For practice, let's walk through the shaping process seen in Example 6-3. At the end of line 3, our traverser is on the vertex for `Sensor 1002688`. Then we walk through all outgoing `send` edges to arrive at any vertex in this sensor's first neighborhood on line 4. The trick here is that a sensor can send information to other sensors or towers. Therefore, we have to prepare for different types of data to process with branching logic in Gremlin.

We would like the result payload to be structured JSON with the following keys: `Label` and `Name`. We create this JSON object and its keys with the `project("Label", "Name")` step. Line 6 fills the `Label` keys in our map with each vertex's label via the `label()` step within a `by()` modulator. Line 7 fills the values for the `Name` key in our map with branching logic via the `coalesce()` step within a different `by()` modulator.

This example of the `coalesce()` step can be be broken down into the following pseudocode:

```
# pseudocode for
# coalesce(values("tower_name"), values("sensor_name"))
    if(values("tower_name") is not None):
        return values("tower_name")
    else:
        return values("sensor_name")
```

`Sensor 1002688` makes for an interesting example in our data because it directly communicates to towers and sensors. Beyond the first neighborhood, however, we can find more paths that connect this sensor to a tower. Let's use the same query as before to examine the second neighborhood of `Sensor 1002688`:

```
sensor = dev.V().has("Sensor", "sensor_name", "1002688"). // look up the sensor
                 next()   // return the sensor vertex
dev.V(sensor).            // look up a sensor
    out("send").          // walk to all vertices in the first neighborhood
    out("send").          // walk to all vertices in the second neighborhood
    project("Label", "Name").   // for each vertex, create a map with 2 keys
      by(label).                // the value for the first key is the label
      by(coalesce(values("tower_name",    // if a tower, return tower_name
                         "sensor_name"))) // else return sensor_name
```

```
{
  "Label": "Sensor",
  "Name": "1061624"
},{
  "Label": "Sensor",
  "Name": "1307588"
},{
  "Label": "Tower",
  "Name": "WhiteCenter"
}
```

These results show that the second neighborhood away from `Sensor 1002688` discovers another tower, `WhiteCenter`. Let's continue walking out and inspect the third neighborhood from `Sensor 1002688`—see Example 6-5 and Example 6-6.

*Example 6-5.*

```
sensor = dev.V().has("Sensor", "sensor_name", "1002688"). // look up the sensor
                 next()  // return the sensor vertex
dev.V(sensor).           // look up a sensor
    out("send").         // walk to all vertices in the first neighborhood
    out("send").         // walk to all vertices in the second neighborhood
    out("send").         // walk to all vertices in the third neighborhood
    project("Label", "Name").     // for each vertex, create a map with 2 keys
      by(label).                  // the value for the first key is the label
      by(coalesce(values("tower_name",    // if a tower, return tower_name
                         "sensor_name"))) // else return sensor_name
```

*Example 6-6.*

```
{
  "Label": "Sensor",
  "Name": "1064041"
},{
  "Label": "Sensor",
  "Name": "1237824"
},{
  "Label": "Sensor",
  "Name": "1237824"
},{
  "Label": "Sensor",
  "Name": "1002688"    // Cycle
},{
  "Label": "Sensor",
  "Name": "1035508"    // Cycle
}
```

Figure 6-15 visualizes all of the data from the first three neighborhoods of `Sensor 1002688` and highlights the cycles in the data with thick edges.

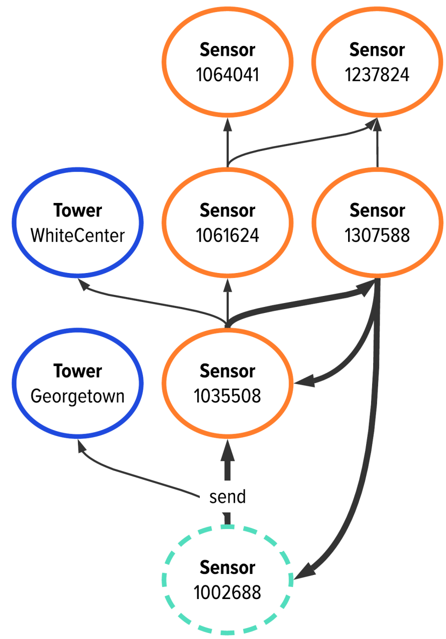



*Figure 6-15. The data reachable in the first three neighborhoods from our starting `Sensor 1002688`*

Figure 6-15 displays the vertices and edges that are within the first, second, and third neighborhoods from `Sensor 1002688`. Close inspection of the bolded edges in Figure 6-15 finds two cycles:

```
1035508 → 1307588 → 1035508
1002688 → 1035508 → 1307588 → 1002688
```

The cycles in our data will be a problem that we will resolve in the next query.

## From This Sensor, What Was Its Path to Any Tower?

Writing multiple Gremlin statements to hardcode the number of steps you walk away from the starting sensor is not an ideal way to write a query. Instead, we want to start at `Sensor 1002688` and explore all communication paths until one of them finds a tower vertex at its root.

We can achieve this with the `until().repeat()` pattern in Gremlin. The use of `repeat()` with `until()` gives you the ability to loop over traversals given some breaking condition. You specify the breaking condition with the `until()` step. If `until()` comes before `repeat()`, it is `while/do` looping. If `until()` comes after `repeat()`, it is `do/while` looping (see Figure 6-16).

| Meaning | Gremlin Syntax | Pseudocode |
|---|---|---|
| do...while | `repeat(traversal).`<br>`until(condition)` | `do{`<br>`this traversal`<br>`}while(condition is true)` |
| while...do | `until(condition).`<br>`repeat(traversal)` | `while(condition is true){`<br>`do this traversal`<br>`}` |

*Figure 6-16. Understanding the `repeat()` step with `until()`*

Example 6-7 shows how to apply this pattern to the idea from Example 6-5 with the `until().repeat()` pattern in Gremlin:

*Example 6-7.*

```
sensor = dev.V().has("Sensor", "sensor_name", "1002688").
                 next()
 dev.V(sensor).                // look up the sensor
     until(hasLabel("Tower")). // until you reach a tower
     repeat(out("send"))       // keep walking out the send edge
```

The query in Example 6-7 will not finish in a timely manner. This is due to the cycles found as you walk from `1002688` up to any tower.

As we saw in Figure 6-15, we want to remove the cycles from our results. There is a step for this in Gremlin: `simplePath()`.

*simplePath()*
: When it is important that a traverser not repeat its path through the graph, the `simplePath()` step should be used. The path information of the traverser is analyzed, and if the path has repeated objects in it, the traverser is filtered.

It really is that…simple.

All we have to do is add the `simplePath()` step within the `repeat()` step pattern. This will insert a filter that eliminates a traverser if its history contains a cycle. Example 6-8 displays the Gremlin code, and Example 6-9 shows the first three results.

*Example 6-8.*

```
1 sensor = dev.V().has("Sensor", "sensor_name", "1002688").
2                  next()
3 dev.V(sensor).                // look up a sensor
4     until(hasLabel("Tower")). // until you reach a tower
5     repeat(out("send").       // keep walking out the send edge
6            simplePath())      // remove cycles
```

*Example 6-9.*

```
{
  "id": "dseg:/Tower/Georgetown",
  "label": "Tower",
  "type": "vertex",
  "properties": {}
},{
  "id": "dseg:/Tower/WhiteCenter",
  "label": "Tower",
  "type": "vertex",
  "properties": {}
},{
  "id": "dseg:/Tower/RainierValley",
  "label": "Tower",
  "type": "vertex",
  "properties": {}
},...
```

The only change from Example 6-7 to Example 6-8 is the use of `simplePath` on line 6. We can see from Example 6-9 that the first three discovered towers are `Georgetown`, `WhiteCenter`, and `RainierValley`. In our application, we want to know more than just which towers were found. We want to know the path from `Sensor 1002688` to the tower.

This brings us to our last Gremlin step and topic for this section: `path()`.

### Using the `path()` step and manipulating its data structure

Let's talk about what the `path()` step in Gremlin does. As you process data in a graph traversal, you are moving around your data. The `path()` step in Gremlin gives you access to the history of where you have been by providing access to all data that has been processed by a traverser.

*path()*
: The `path()` step (map) examines and returns the full history of a traverser.

This is roughly like leaving breadcrumbs around your graph as you move from place to place.

We introduce the `path()` step in Example 6-10 and display the results in Example 6-11.

*Example 6-10.*

```
1 sensor = dev.V().has("Sensor", "sensor_name", "1002688").
2                  next()
3 dev.V(sensor).
4     until(hasLabel("Tower")). // until you reach a tower
5     repeat(out("send").       // keep walking out the send edge
6            simplePath()).     // remove cycles
7     path().  // all objects will be towers; get their full history
8        by(coalesce(values("tower_name",    // if the vertex in the path is a tower
9                           "sensor_name"))) // else the value from a sensor vertex
```



In the path data structure, `labels` is *not* the same as a vertex label or an edge label.

Let's walk through the new steps of Example 6-10. As before, lines 1 through 6 start at a sensor and walk through the `send` edges to any tower, considering only noncyclic paths. Then, for all reachable towers, the `path()` step on line 7 asks each traverser for its full path through the data. Line 8 uses a `by()` modulator to indicate how we want to see that data: we want to see the `tower_name` if the vertex is a tower, or else we want to see the `sensor_name`.

Example 6-11 shows the first three results of Example 6-10. We see two of the paths we drew in Figure 6-15.

*Example 6-11.*

```
{
  "labels": [[],[]],
  "objects": ["1002688", "Georgetown"]
},{
  "labels": [[],[],[]],
  "objects": ["1002688", "1035508", "WhiteCenter"]
},{
  "labels": [[],[],[],[]],
  "objects": ["1002688", "1035508", "1061624", "1237824", "RainierValley"]
},...
```

The results in Example 6-11 show three different ways in which you can arrive at towers by starting from `Sensor 1002688`. The first two paths confirm what we discovered as we walked through the first and second neighborhoods of `1002688`; we just

see the data in a different structure: `["1002688", "1035508", "WhiteCenter"]`. This notation means the following path was found in the traversal:

```
1002688 → 1035508 → WhiteCenter
```

More than a thousand different ways to walk from `Sensor 1002688` to a tower vertex are shown in the accompanying Studio Notebook.

When you use `path()` there are two things you must understand deeply: how to assign labels with `as()` and how to shape the results with `by()`. Let's go through each of these topics in detail.

**How to assign labels with as().** There are two keys to the `path()` data structure: `labels` and `objects`. A label is created for a path object with the `as()` step. Essentially, you are assigning a variable name to the data you are processing in your path. We didn't use the `as()` step in the first version of our query, so the `labels` key in the result payload in Example 6-7 contained no data.

Let's use the `as()` step now to assign variable names to our path data structure in Example 6-12, and then we'll reinspect the resulting payload in Example 6-13.

*Example 6-12.*

```
sensor = dev.V().has("Sensor", "sensor_name", "1002688").
                 next()
dev.V(sensor).
      as("start").           // label 1002688 as "start"
    until(hasLabel("Tower")).
    repeat(out("send").
             as("visited").  // label each vertex on the path as "visited"
           simplePath()).
    as("tower").             // label the end of the path as "tower"
     path().
       by(coalesce(values("tower_name",
                          "sensor_name")))
```

*Example 6-13.*

```
{
  "labels": [["start"], ["visited", "tower"]],
  "objects": ["1002688", "Georgetown"]
},{
  "labels": [["start"], ["visited"], ["visited", "tower"]],
  "objects": ["1002688", "1035508", "WhiteCenter"]
},{
  "labels": [["start"], ["visited"], ["visited"], ["visited", "tower"]],
  "objects": ["1002688", "1035508", "1061624", "1237824", "RainierValley"]
},...
```

Example 6-12 shows how the `as()` step introduces values within the `labels` key of the `path()` data structure. The values within `labels` and `objects` have a 1:1 mapping. Let's look again at the second example from Example 6-13 to understand how the labels map to the path:

```
{
  "labels": [["start"], ["visited"], ["visited", "tower"]],
  "objects": ["1002688", "1035508", "WhiteCenter"]
}
```

1. The value `["start"]` maps to `1002688`
2. The value `["visited"]` maps to `1035508`
3. The value `["visited", "tower"]` maps to `WhiteCenter`

We can confirm this mapping by looking back at our query in Example 6-12. We labeled the starting sensor with `as("start")`. Each vertex that was accessed within the `repeat(out("send"))` step was labeled with `as("visited")`. Last, only towers are passed to line 9 due to the conditional filter from line 5: `until(hasLabel("Tower"))`. Therefore, any tower vertex will receive a second label from line 9 with `as("tower")`.

Using `as("<some_label>")` is powerful because we are able to use the `path()` step's data structure to provide specificity to the resulting payload.

There is one last concept to detail about using `path()` before we move on to other queries.

**How to shape `path()` results with `by()`.** The use of `by()` in Example 6-12 allows you to perform an operation, or another step, to each object in the path. In our example, we wanted to return the primary key for each vertex in the path. However, the vertex's label could be a tower or a sensor. Therefore, we added a condition within the `by()` modulator to process tower vertices one way and sensor vertices another way.

When formatting the elements of `path()`, the `by()` modulators in Gremlin are applied in a round-robin fashion, meaning they are applied to the traversal objects in a cyclical order. In a case in which there are two `by()` steps:

1. The first `by()` step operates on the first traversal object
2. The second `by()` step operates on the second traversal object
3. Back to the first `by()` step for the third traversal object
4. Back to the second `by()` step for the fourth traversal object
5. And so on…

In the example here, all of the objects in the path were vertices, so we needed to create only one `by()` modulator to handle vertex objects. You will see examples in the next chapter in which we need multiple `by()` modulators because we are processing both vertices and edges in our path's data structure.

### From Bottom Up to Top Down

All of the queries and code in this section were designed to teach you how to walk from leaves to roots in a hierarchical graph. You can think of this as walking from the bottom of your tree to its top.

Once at the top, you may want to walk back down. So let's next explore how to start at a tower and walk down to the sensors connected to it and the various concepts you will encounter along the way.

The upcoming examples build up to a question that we cannot resolve with the information we have. We designed this experience on purpose to set the stage for the production tips in Chapter 7.

## Querying from Roots to Leaves in Development

Edge Energy has to maintain an understanding of its network's topology. It needs to know, at all times, which sensor's data an individual tower is processing.

Knowing which sensors connect to a particular tower helps answer two important questions about this dynamic communication network. It helps Edge Energy understand whether a specific tower is overloaded or underutilized. We are going to help Edge Energy understand its network by answering the following questions in this section:

1. First, we need to find an interesting tower to explore in our data.
2. Which sensors have directly connected to that tower?
3. From that tower, find all sensors that have connected to it.

Our example data has the ability to answer the questions for these scenarios. However, we do not have enough information to answer the whole scope of question 3—just part of it. Figuring out how to answer question 3 in its entirety will be the purpose of Chapter 7.

Let's continue to develop our queries and knowledge of the Gremlin query language with our first question.

## Setup Query: Which Tower Has the Most Sensor Connections So That We Could Explore It for Our Example?

The first thing we want to do is find a tower that has interesting connectivity in our graph.

Why are we doing this right out of the gate?

When we start playing with new data, we run a couple queries to understand it better. Keep in mind, this is not something we would put in production. This is something we needed to do for educational purposes to find interesting data to work with.

So that we can find an interesting tower to work with, we will want to process all towers in our graph and then order the towers according to the number of incoming edges. Then we want the primary key of the tower with the highest degree. Let's take a look in Example 6-14 at the Gremlin query that achieves this.

> The query in Example 6-14 is for exploration and development purposes only. It is expensive to run in a distributed system because it performs a full table scan of the towers and of each tower's edge table.

*Example 6-14.*

```
1 dev.V().hasLabel("Tower").          // for all towers
2         group("degreeDistribution").// create a map object
3           by(values("tower_name")). // the key for the map: tower_name
4           by(inE("send").count()).  // the value for each entry: its degree
5         cap("degreeDistribution").  // barrier step in Gremlin to fill the map
6         order(Scope.local).         // order the entries within the map object
7           by(values, Order.desc)    // sort by values, decreasing
```

In Example 6-14, we construct a map that represents the degree distribution of the tower vertices in our graph. The `group()` step on line 2 creates a map object called `degreeDistribution`. We need to follow the `group()` step with definitions for the map's keys and values. The `by()` modulator on line 3 defines that `tower_name` will be the key for any entry in this map. Line 4 indicates that the value associated to a specific `tower_name` will be the total number of incoming edges to that tower.

Line 5 introduces a new concept in Gremlin—barrier steps:

*Barrier steps*
: Barrier steps force the traversal pipeline to complete up until that point before continuing.

The use of `cap()` on line 5 in Example 6-14 is an example of a barrier in Gremlin. Here, `cap()` iterates the traversal up until that step and passes the object with the name `degreeDistribution` into the next step in the pipeline. We mentioned in the last chapter that local scope orders elements within an object, whereas global scope would order all objects in a traversal pipeline. We see this in action again in line 6; `order(Scope.local)` orders the elements within the map object `degreeDistribution`.

Finally, line 7 in Example 6-14 provides the rule for this ordering: we want descending order according to the values in the map. A sample of the results is:

```
{
  "Georgetown": "7",
  "WhiteCenter": "7",
  "PioneerSquare": "6",
  "InternationalDistrict": "6",
  "WestLake": "5",
  "RainierValley": "5",
  "HallerLake": "4",
  "SewardPark": "4",
  "BeaconHill": "4",
  ...
}
```

We found a few useful towers, so let's pick one. We see Georgetown has seven sensors; let's determine which ones connected directly to that tower.

## Which Sensors Have Connected Directly to Georgetown?

We'll start by querying a tower and the sensors that have directly connected to it. We can follow the same pattern we did in Example 6-12 when we were working from sensors:

```
sensor = dev.V().has("Sensor", "sensor_name", "1002688").next()
dev.V(sensor).
    out("send").
    project("Label", "Name").
      by(label).
      by(coalesce(values("tower_name", "sensor_name")))
```

This time, we want to start at a tower and access its incoming communication from sensors. We can change the query in Example 6-12 to the query shown in Example 6-15. The results of this query follow.

*Example 6-15.*

```
tower = dev.V().has("Tower", "tower_name", "Georgetown").next() // get Georgetown
dev.V(tower).                      // look up Georgetown
    in("send").                    // traverse in to sensors
```

```
project("Label", "Name").      // create a map with two keys
  by(label).                   // of the values for "Label"
  by(values("sensor_name"))    // the values for "Name"
```

```
{
  "Label": "Sensor",
  "Name": "1002688"
},{
  "Label": "Sensor",
  "Name": "1027840"
},{
  "Label": "Sensor",
  "Name": "1306931"
},...
```

The results of Example 6-15 show that `Sensor 1002688` connected to `Georgetown`, a result we expected to see. Even though we didn't show all seven in this text, the full results in the Studio Notebook show that `Georgetown` has seven sensors that directly connected to it.

Edge Energy needs to know all sensors that use this tower for communication. We already know that `Sensor 1002688` has an incoming edge from `1307588`. This leads us to ask, how many other sensors are using the network to send their information to `Georgetown`?

To answer that question, we will want to walk recursively from this tower through all incoming edges until we have found all sensors in this tree of communication. The next and last section of this chapter applies the use of `repeat()/until()` from the last section to walk from this tower down to all sensors.

## Find All Sensors That Connected to Georgetown

We have been working through querying through our data for a few sections. This last question is the final query needed to answer the question of our larger, complex problem: what happens if a tower fails?

The logical way to approach this last query won't actually work, but we are going to show it to you anyway because it is the logical next step that everyone tries; we see it all the time. We encourage learning through trying logical next steps, so that is exactly what we are about to do.

It is very common to find patterns of working Gremlin queries and apply them to new problems; this is the pattern we are talking about that will lead to a faulty solution to our new question.

Let's take a look back at how we recursively walked up from sensors to towers:

```
dev.V(sensor).                // look up a sensor
    until(hasLabel("Tower")). // until you reach a tower
    repeat(out("send").       // keep walking out the send edge
           simplePath())      // remove cycles
```

The logical next step is to transform that query to do exactly the reverse: walk from towers to sensor. Let's apply the same pattern but switch the type of objects we start and end with. In Example 6-16, we start with towers and recursively walk to sensors.

*Example 6-16.*

```
tower = dev.V().has("Tower", "tower_name", "Georgetown").next() // get Georgetown
dev.V(tower).                 // look up a tower
    until(hasLabel("Sensor")). // until you reach a sensor
    repeat(__.in("send").      // need to use the Anonymous traversal: __.
           simplePath())       // remove cycles
```

Example 6-16 requires a new step in Gremlin called the Anonymous traversal. In Groovy, `in()` is a reserved keyword, and DataStax Studio uses the Groovy variant of Gremlin for developing traversals. Therefore, the `in()` Gremlin step must be prefixed with the Anonymous traversal for our example. The full result payload is shown in Example 6-17.

> The Anonymous traversal `__.` is used to resolve many variants of Gremlin that have clashes with reserved language-specific keywords such as `in`, `as`, or `values`. Refer to the Apache TinkerPop documentation for specifics within your coding language of choice.

*Example 6-17.*


```
{
  "id": "dseg:/Sensor/1002688",
  "label": "Sensor",
  "type": "vertex",
  "properties": {}
},{
  "id": "dseg:/Sensor/1027840",
  "label": "Sensor",
  "type": "vertex",
  "properties": {}
},{
  "id": "dseg:/Sensor/1306931",
  "label": "Sensor",
  "type": "vertex",
  "properties": {}
}...
```

Wait a second. Inspecting the full result payload in Example 6-17 reveals that the query from Example 6-16 found only the same seven sensors from the tower's first neighborhood.

This is not what we want.

The query in Example 6-16 does not give us what we want because it has a stopping condition of any sensor vertex on line 2 with `until(hasLabel("Sensor"))`. Instead, we want to recursively walk any depth until we find all sensors. Let's remove this condition and try again:

```
tower = dev.V().has("Tower", "tower_name", "Georgetown").next() // get Georgetown
dev.V(tower).                  // look up a tower
    repeat(__.in("send").      // keep walking in the send edge
           simplePath())       // remove cycles
```

If you ran this second version of our query in DataStax Studio, you most likely saw the error in Table 6-5:

*Table 6-5. An example of a system error due to a traversal taking longer than 30 seconds*

| System error |
| --- |
| Request evaluation exceeded the configured threshold of realtime_evaluation_timeout at 30000 ms for the request |

At the heart of this error is the trouble of recursively walking through trees in a graph. We were starting at the root of a tree and completing a full search down to all the leaves in the tree.

This is extremely expensive.

## Depth Limiting in Recursion

There are many ways to address the error in Table 6-5. One way is to limit how deep a traverser travels away from its starting point.

You control the number of times a traverser executes a loop with the `times(x)` step. The pattern `repeat(<traversal>).times(x)` is one of the most popular ways to limit the depth of a recursive traversal in Gremlin. In this pattern, the value `x` tells a traverser to perform the repeat loop `x` number of times.

In the following query, we show `repeat(<traversal>).times(3)`. This means that from a tower, a traversal walks out only three `in()` edges and then stops:

```
tower = dev.V().has("Tower", "tower_name", "Georgetown").next() // Georgetown

dev.V(tower).               // look up Georgetown
    repeat(__.in("send"). // repeat walking in the send edge
           simplePath()). // remove cycles
```

```
    times(3).              // repeat only 3 times total
    path().                // get the path
      by(coalesce(values("tower_name",   // if a tower, return tower_name
                         "sensor_name")))// else, return the sensor name
```

The results are:

```
{
  "labels": [[],[],[],[]],
  "objects": ["Georgetown","1235466","1257118","1201412"]
},{
  "labels": [[],[],[],[]],
  "objects": ["Georgetown","1290383","1027840","1055155"]
},{
  "labels": [[],[],[],[]],
  "objects": ["Georgetown","1235466","1059089","1255230"]
},...
```

The benefits of depth limiting with our example data is that we can now perform part of our final query for this chapter. However, we reduced the scope of the question from finding all sensors to only those sensors within a specific depth, namely 3. There are many more reachable sensors that we are missing by limiting depth.

To find them all, we need to revisit time in our example data.

# Going Back in Time

We realize we left you hanging with that last query.

We set up the need to walk from a root (a tower) down to all leaves (sensors), and it didn't work as expected. All that is to say, your journey as a data engineer for Edge Energy isn't over quite yet. We will keep using Edge Energy's example as we transition into the next chapter where we explain how to adjust our setup to answer our question.

Let's travel deeper into the structure of our trees to find the branches that get us out of this forest of problems. We are going to teach you how to prune the data you process in your query by limiting your branching factor, limiting by depth, and removing cycles. And if your eyes aren't rolling at the terrible puns by now, just wait.

CHAPTER 7

# Using Trees in Production

Whether you are modeling corporate structures or unbounded networks of IoT sensor communication, hierarchical data fits very well into graph technologies.

As we see it, especially with unbounded and hierarchical data, the mental distance between the data on disk and using it is much shorter when you use graph technology. However, as we saw at the end of the previous chapter, simple questions with expressive languages and natural models can open the door to unexpected behavior.

Namely, it is easy to think about starting at the root of a tree and walking all the way down to its leaves. And graph technologies enable the code for this to be quite simple.

However, the simplicity that comes with reasoning about complex, tree-structured problems obfuscates the complexity of processing the data's natural hierarchical structure.

## Chapter Preview: Understanding Branching Factor, Depth, and Time on Edges

This chapter will have four main sections. Each section builds upon the previous one to walk through modeling the `time` property on edges to resolve our error at the end of Chapter 6.

In the first section, we'll build upon the data introduced in the last chapter by adding two complexities: time on edges and valid paths. The second section delves into why a valid communication tree reduces the amount of data to process. We will update and walk through the production version of our graph schema in this section. The third and fourth sections of this chapter revisit the same set of queries from the last chapter. This time, however, we will apply our knowledge of valid trees and the new production schema to significantly reduce the amount of data processed in each query.

At the end of this chapter, you will have everything you need to start working with trees in your own data. We consider the content in Chapters 6 and 7 to contain a streamlined yet complete introduction to working with hierarchical structured data in a production application with graph technologies.

To help get you there, let's go back to the data we created for this example and follow the edges throughout time.

## Understanding Time in the Sensor Data

The data we created and introduced in "Understanding Hierarchies with Our Sensor Data" on page 162 simulates how sensors send data to each other and to cell towers. We introduced this data within the context of a power company, Edge Energy. The data engineers at Edge Energy have to build a system capable of reporting sensor coverage in the event of a tower failure.

This bring us to the concept of time in our data. The sensors collect and send data throughout the network at specific time intervals. This means that the number of vertices in our graph will be fixed, and it is the relationships in the graph that grow over time.

We model the dynamic communication over time intervals with a `timestep` property on the edges. Let's look at our data in Figure 7-1 to see how time is part of the communication network.

The only difference between the examples in Chapter 6 and Figure 7-1 is the inclusion of time on the edges.

Consider the `Seattle` tower at the bottom of Figure 7-1. `Sensor S`, to the lower right of `Seattle`, has an edge with the values `[0,3]`. In our application, this means that the sensor sent information to the `Seattle` tower at `timestep 0` and `timestep 3`. In other words, this sensor directly connected with the `Seattle` tower twice. You can also see that `Sensor S` connected with a nearby neighbor at `timesteps` `1`, `2`, `4`, and `5`.

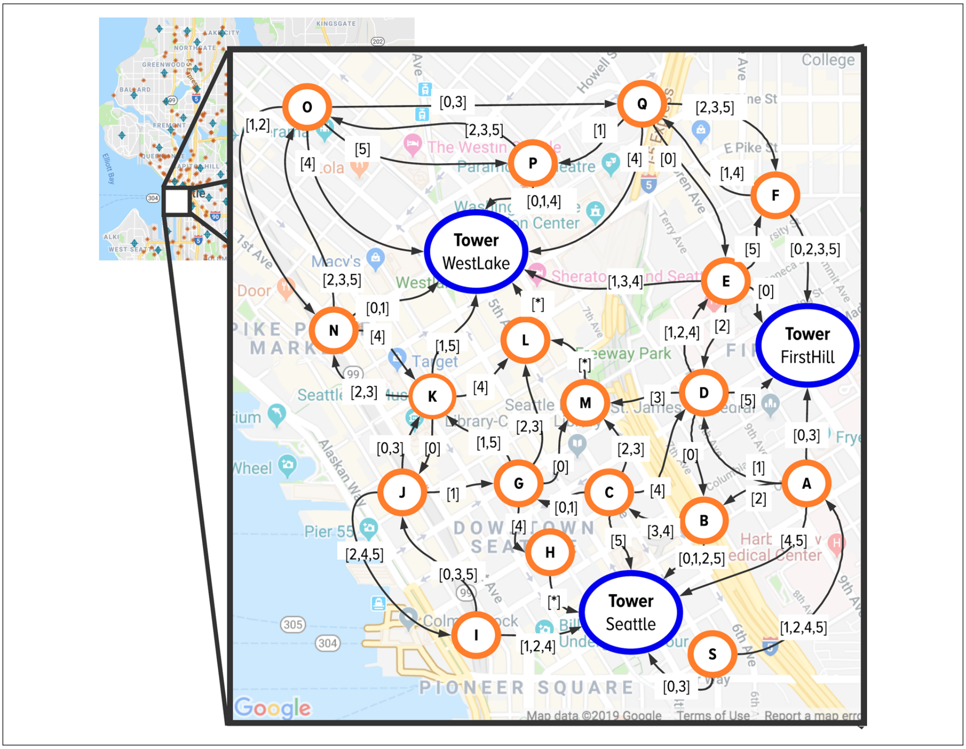



*Figure 7-1. Our first glimpse into how a `timestep` property on edges augments the communication network for this chapter's example*

To understand how to use time on our edges in our upcoming queries, we need to introduce four topics. These four topics will be the next four sections:

1. Understanding time from the bottom up
2. Valid paths from the bottom up
3. Understanding time from the top down
4. Valid paths from the top down

Let's start with showing you how to walk through the data from sensors up to towers.

### Understanding time in hierarchies of data: From the bottom up

To help you understand how to use time, consider it in context. Recalling our setup from Chapter 6, the first queries walk from the leaves up to the root. This is a walk from the sensors to the tower that received their data.

*Every* walk from a sensor to a tower is no longer a *valid* walk because we have to consider the timing of the communication along the way. That is, a message passed from a sensor at `timestep 3` will be passed along from its recipient at `timestep 4`. Let's look at an example; we'll start by zooming in on valid walks from `Sensor S` to nearby towers in Figure 7-2.

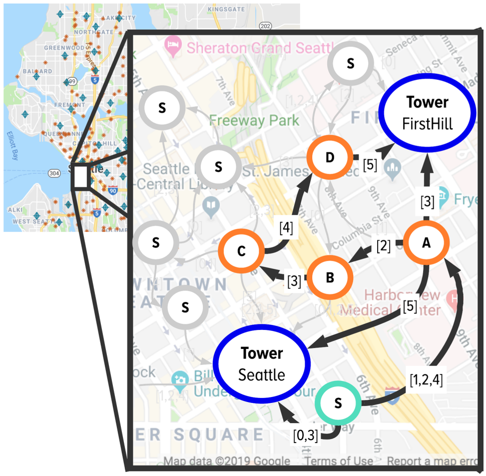



*Figure 7-2. Zooming in on valid walks from `Sensor S` to nearby towers*

The examples in Figure 7-2 show five valid paths from `Sensor S` to towers. Let's walk through two scenarios and then show you where to start to find the other three.

The first valid path is to follow the first message sent by `Sensor S` at `timestep 0`. This walk goes from `Sensor S – 0 → Seattle`. This one is pretty easy.

For a more complicated example, let's follow the second communication path that leaves from `Sensor S`. The second path starts at `timestep 1`. Here we find a much deeper path. This walk goes from:

```
Sensor S – 1 → A
       A – 2 → B
       B – 3 → C
       C – 4 → D
       D – 5 → FirstHill
```

When we are following these paths, we walk through time by incrementing by one along the way. You can keep walking through the other valid paths from Figure 7-2. The third valid path starts at `timestep 2`, the fourth valid path starts at `timestep 3`, and the last one starts at `timestep 4`.

In Chapter 6 we saw seven paths from `Sensor S`, but now we know that two of them weren't possible!

We also can flatten the data and examine these paths in a more hierarchical form. Let's look at the hierarchy from `Sensor S` to nearby towers in Figure 7-3.

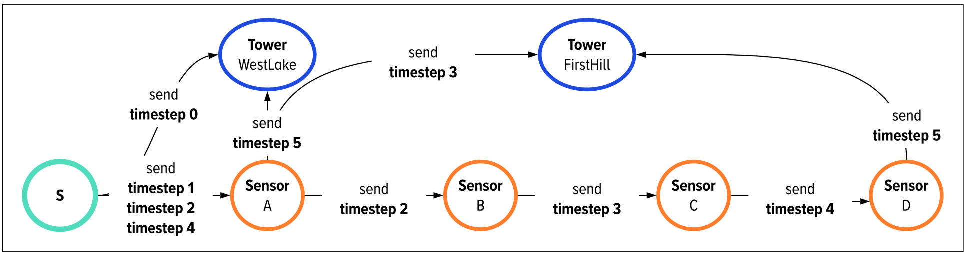



*Figure 7-3. Understanding the hierarchy of communication from `Sensor S` to any nearby tower throughout time*

Figure 7-3 shows the same data as Figure 7-2 but in a more hierarchical form. It may be easier to see the four unique paths of Figure 7-3 by counting the edges that go into the tower vertices. Here, we can also see the other paths we didn't cover from Figure 7-2. We see the path of:

```
Sensor S – 2 → A
       A – 3 → FirstHill
```

Whichever mental model you prefer, we are digging into how these trees work for a few reasons. First and most important, the dynamic nature of connections and communication between the vertices in this dataset represents a real-world scenario for how devices in the field transmit their data back to databases.

Second, the use of time on edges sets us up to understand what type of communication tree would be observable in the real world. Let's take a look at valid and invalid paths throughout time when walking up from sensors to towers.

#### Valid and invalid paths from the bottom up

Let's start by thinking about how to correctly interpret time when you follow it from a sensor to a tower.

Conceptually, you can think of a valid path from a sensor to a tower as passing the data on to the next sensor in order. In the data, a valid path increments time by one as you walk through the edges.

Continuing on conceptually, an invalid path is when you try to pass information to another sensor out of order. This is like missing your train: you got there either too late or too early.

To put this into practice, let's explore examples of valid and invalid paths. First, Figure 7-4 shows an invalid path; it's invalid because the sensor's data was received late.

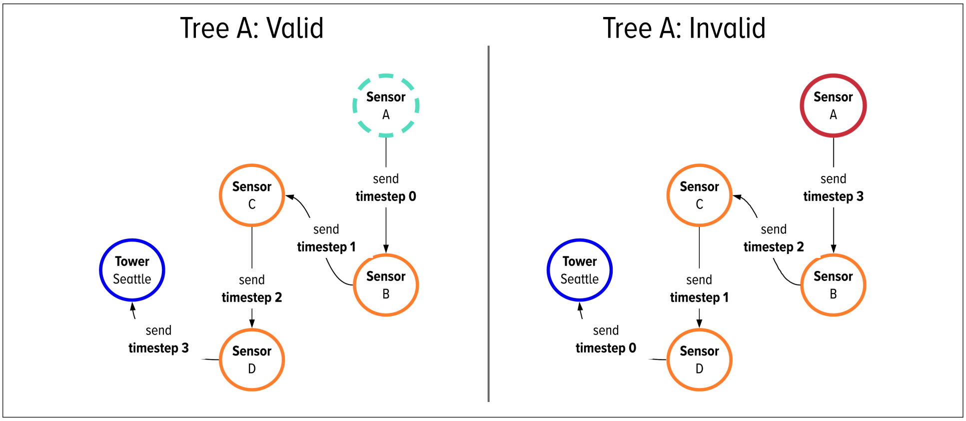



*Figure 7-4. An example of valid (left) and invalid (right) paths from `Sensor A` to `Seattle`*

Figure 7-4 shows two paths from `Sensor A` to the `Seattle` tower. The path on the left is valid because time on the edges correctly increments by one along the way. The path on the right is invalid because each exchange with a sensor is out of order. `Sensor A` sends its information to `Sensor B` after `Sensor B` has communicated with `Sensor C`. The same problem happens with every exchange along the path on the right in Figure 7-4.

Let's take a look at another type of invalid path in Figure 7-5; the paths on the right of Figure 7-5 show instances of sensors communicating too early.

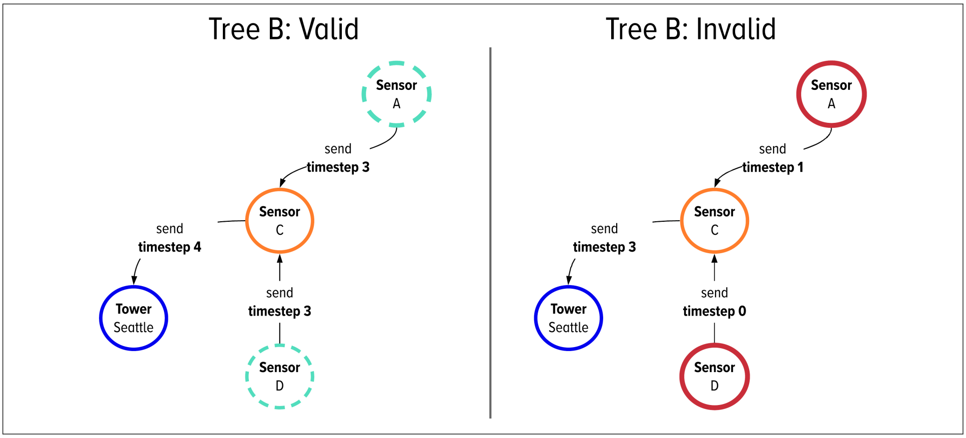



*Figure 7-5. A second example of valid (left) and invalid (right) paths to Seattle*

Figure 7-5 shows paths from `Sensor D` and `Sensor A` to the `Seattle` tower. On the left, the paths are valid. `Sensor D` sends its data to `Sensor C` at `timestep 3`, as does

Sensor A. Then Sensor C collects all the data and sends it on to the Seattle tower at timestep 4.

The paths on the right in Figure 7-5 are invalid. Sensor D sends its data to Sensor C at timestep 0; the data for Sensor D leaves Sensor C at timestep 1 (not shown). Sensor A sends its data to Sensor C at timestep 1; the data for Sensor A leaves Sensor C at timestep 2 (not shown). Figure 7-5 shows that Sensor C communicated with the Seattle tower at timestep 3. This means that the data for D and A was not a part of that communication because it was passed along different paths at timestep 1 and timestep 2, respectively.

That covers everything we need to know about our data when we walk from leaves to roots. Let's look at how we apply time in the reverse direction.

#### Understanding time in hierarchies of data: From the top down

The last concept for our example applies time as we walk from towers down to all sensors. These paths represent how we would figure out which sensors connected to a tower at a certain time.

The key here is that a valid path from a tower down to a sensor has to follow time in decreasing order, exactly by 1.

To see this, let's zoom in and examine the network that sends its information to the WestLake tower in Figure 7-6. Figure 7-6 is dense with information. To understand what it is showing, we recommend starting with what you know how to trace by following valid paths from sensors up to the tower. Starting this way makes it much easier to accomplish our ultimate goal: walking in reverse from WestLake down to sensors.

Let's start with Sensor M, at lower right in Figure 7-6. We want to follow the valid path from the sensor up to WestLake:

```
Sensor M – 2 → I
       I – 3 → F
       F – 4 → WestLake
```

The goal is to be able to see this in reverse from WestLake back to Sensor M. So trace that same path but in the opposite direction:

```
WestLake - 4 → F
       F - 3 → I
       I - 2 → Sensor M
```

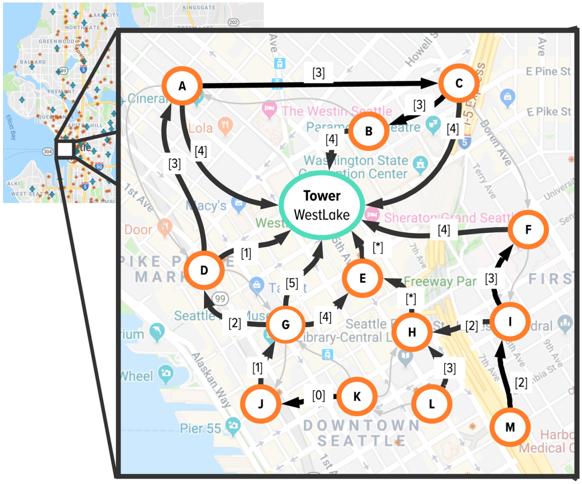



*Figure 7-6. Zooming in on valid walks from `WestLake` to all sensors that connect to it*

Let's unroll all valid paths that arrive at `WestLake` at `timestep 4`. The hierarchy from the root, `WestLake`, down to all sensors that connected to it is shown in Figure 7-7. All paths from Figure 7-7 can be found in Figure 7-6. We just untangled their representation on the map to look at their hierarchical structure.

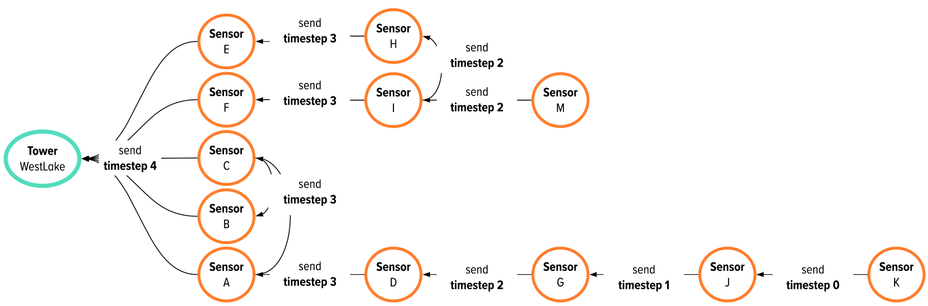



*Figure 7-7. Understanding the hierarchy of communication received by the `WestLake` tower from any sensor at `timestep 4`*

It is easier to walk backwards from towers to sensors in the hierarchical structure shown in Figure 7-7. For example, follow the path backwards from the `WestLake` tower to `Sensor M` in this image. The path is the same as before, but it is easier to see how time decreases in this image.

We find it easier to see the paths when you look at the hierarchical structure of the data as shown in Figure 7-7. But you may prefer to follow them according to their geo-location, like in Figure 7-6.

As long as you can see how to decrease time as we walk from a tower back down to sensors, then we have achieved our goal.

Before we can update our production schema, we have one last concept to understand: valid paths from roots to leaves.

### Valid and invalid paths from the top down

Think about what we are doing when we find a valid path from a tower down to all sensors. We have reversed the process we used for walking up from sensors to a tower.

In this reversal, we walked backwards in time. Specifically, we decreased the `timestep` values on the edges by one along the way.

Let's look at another side-by-side example of valid and invalid paths. This time, however, we are considering the perspective from the tower down to the sensors, back through time. The communication path on the right in Figure 7-8 is invalid because communication was too late or too early along the path.

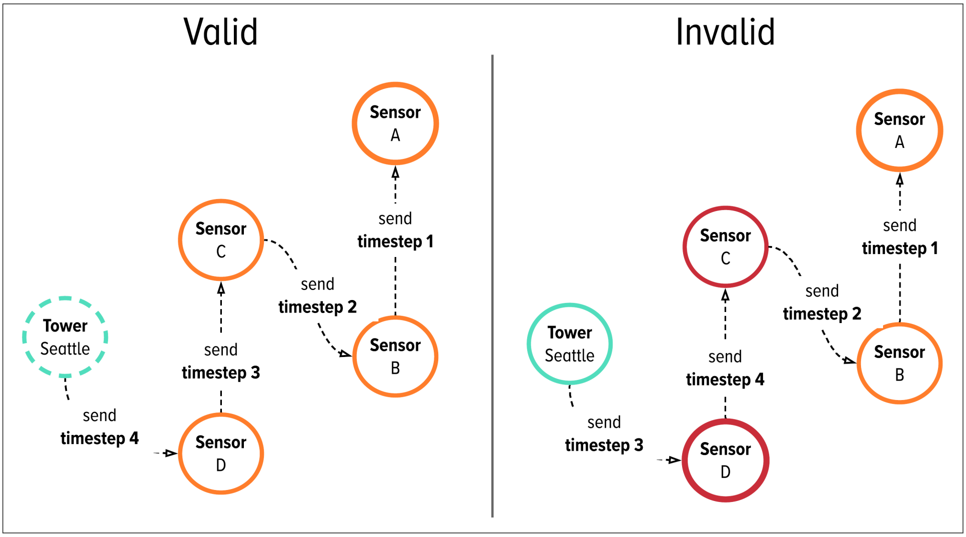



*Figure 7-8. An example of valid and invalid paths from a tower*

Figure 7-8 shows paths from the `Seattle` tower to `Sensor A`. On the left, the path is valid:

```
Seattle - 4 → D
      D - 3 → C
      C - 2 → B
      B - 1 → A
```

Contrast the path on the left with its invalid representation on the right. The path on the right is invalid because of the time at which `Sensor D` received its information:

```
Seattle - 3 → D (too late for the next connection)
      D - 4 → C (too early for the next connection)
      C - 2 → B
      B - 1 → A
```

`Seattle` sent its data to `Sensor D` too late; `Sensor D` had already passed its data to `Sensor C`. The communication path is also invalid between `D` and `B`.

It is much harder to reason about time when going backwards. The trick here is that a valid path from a tower down to a sensor has to follow our property in decreasing order, exactly by 1.

## Final Thoughts on Time Series Data in Graphs

Understanding time in this dataset is easy for modeling: we added it to our edges. We will see the production schema in a coming section.

The detail and difficulty come in when we want to use time in our queries. Aside from all of the images and detail in this chapter, using time in this example boils down to the following tip:

> **Rule of Thumb #11**
>
> Time goes up on the way up, and time goes down on the way down. When this rule of thumb isn't true, the path is invalid and should be filtered out of the results.

Now that we know how to use time on our edges, let's explain why this resolves our error from Chapter 6. We want to limit our results to valid paths, and therefore, we are mitigating our graph's branching factor.

Let's explain what branching factor is and why you need to know about it for this example and others.

# Understanding Branching Factor in Our Example

We ended Chapter 6 with a problem. We were unable to walk from a tower down to all sensors that connected to it because of the data's branching factor.

Let's dig into the details of this concept and illustrate the processing complexity that comes with walking through highly branching data.

## What Is Branching Factor?

Branching factor is what happens when you walk from one vertex through relationships to many other vertices. Formally, we define branching factor as follows:

*Branching factor*
: A graph's branching factor (BF) is the expected, or average, number of edges for any vertex.

You can think of this as splitting one process, or one traverser, into many. We illustrate this in Figure 7-9.

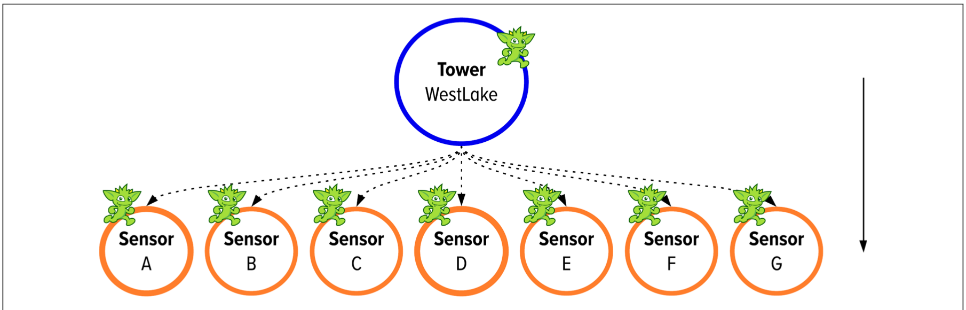



*Figure 7-9. An example of `WestLake`'s branching factor*

In Figure 7-9, the `WestLake` tower vertex has seven edges adjacent to seven unique vertices. We say the `WestLake` tower has a branching factor of 7.

Your data's branching factor affects traversal performance. For example, a traversal that starts at the `WestLake` tower creates one traverser in the pipeline. When you walk from the `WestLake` tower to all incoming vertices, the single traverser splits for every possible edge. We end up with seven total traversers on the sensor vertices, shown at the bottom of Figure 7-9.

The processing overhead for a traversal correlates to a graph's branching factor. Roughly, the number of traversers maps to the number of threads required to execute a traversal. You can calculate the number of threads required to process a query map with the equation shown in Figure 7-10.

$$\sum_{n \geq 0} (BF)^n$$

*Figure 7-10. The computational overhead for a traversal according to its depth, `n`, and the graph's branching factor, `BF`*

That sounds great and all, but why should you care?

Let's say your graph's expected branching factor is 3. Starting at a single vertex, you have 1 traverser. Walking one neighborhood away creates 3 traversers. Two neighborhoods away creates 9; three neighborhoods away creates 27. When you are four neighborhoods away, you are processing 81 traversers, just for that level. The total number of traversers you have created is 1 + 3 + 9 + 27 + 81 = 121.

The exponential growth can quickly get out of hand. Figure 7-11 shows just how quickly.

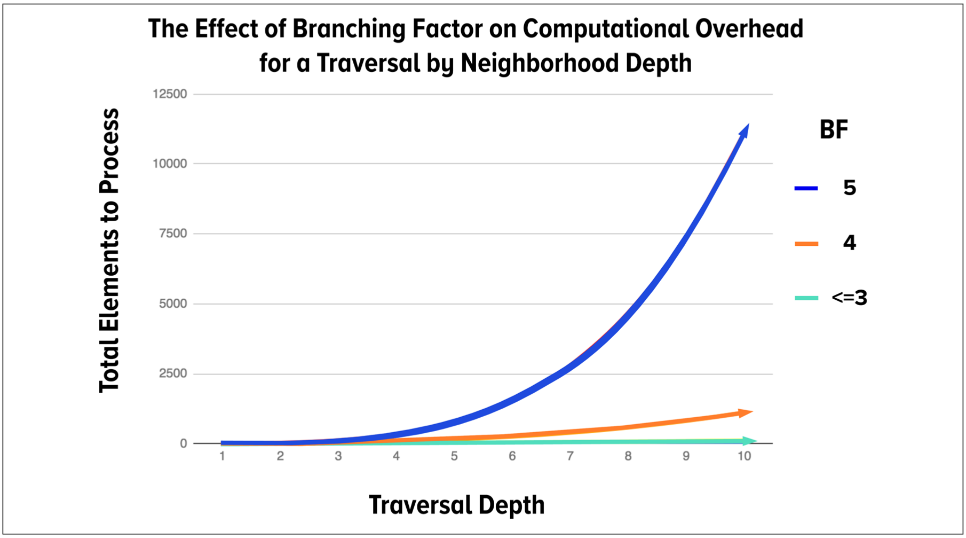



*Figure 7-11. Looking at the total pieces of data needed to process a traversal according to the traversal's depth, and the graph's branching factor*

The message from Figure 7-11 is that a graph's branching factor yields exponential growth on the amount of data that you have to process as you explore multiple neighborhoods of data. Loosely speaking, you can equate one Gremlin traverser to one thread in your computer. This means that the number of threads required to explore your data grows exponentially.

## How Do We Get Around Branching Factor?

The beauty of working with graph data in Apache Cassandra is that we already have all the tools necessary to tame your data's branching factor. A primary way to mitigate a query's branching factor goes back to how you store your data on disk.

One of the best tips we can offer is to use properties on edges to give yourself a way to navigate your data's branching factor during queries.

**Rule of Thumb #12**

Cluster your edges on disk so that you can sort through them in your queries and mitigate the effect of your data's branching factor.

Let's apply our understanding of branching factor. We want to update our development schema so that our production queries are less affected by our tree's branching factor.

## Production Schema for Our Sensor Data

Our new understanding of time on edges and our exploration in development give us two optimizations for our production schema. First, our new understanding of time on edges, valid paths, and branching factor indicates *why* we need to cluster our edges by time. Second, our queries in Chapter 6 illustrated that we will be traversing the `send` edge in both directions. Therefore, our second change will be to add a materialized view on the `send` edge labels for bidirectional usage in traversals.

Figure 7-12 illustrates a production version of our conceptual data model with these changes.

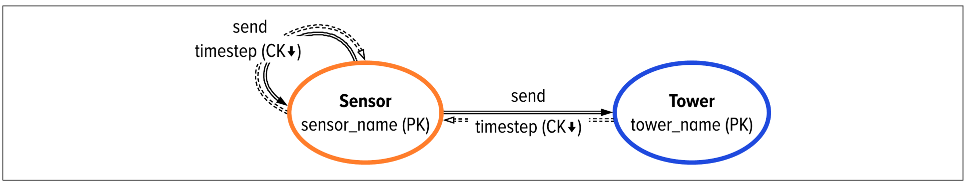



*Figure 7-12. Our production schema model for the final set of tree queries in this chapter*

In Figure 7-12, the use of materialized views on each `send` edge is indicated by a dotted line going in the reverse direction. We also see that our edges will be clustered by time, decreasing with the `timestep (CK↓)` notation.

Applying the Graph Schema Language (GSL), we cluster our edges by time with:

```
schema.edgeLabel("send").
       ifNotExists().
       from("Sensor").
       to("Sensor").
       clusterBy("timestep", Int, Desc).
       create()
```

```
schema.edgeLabel("send").
       ifNotExists().
       from("Sensor").
       to("Tower").
       clusterBy("timestep", Int, Desc).
       create()
```

We create these indexes in our schema code with:

```
schema.edgeLabel("send").
       from("Sensor").
       to("Sensor").
       materializedView("sensor_sensor_inv").
       ifNotExists().
       inverse().
       create()

schema.edgeLabel("send").
       from("Sensor").
       to("Tower").
       materializedView("sensor_tower_inv").
       ifNotExists().
       inverse().
       create()
```

The edge label syntax in the preceding code creates materialized views for each respective `send` edge label. By using the `inverse()` convenience method, we are applying the same order to the edges in the reverse direction. This means that the edges will have a clustering key of `timestep` in the reverse direction.

**Bonus Rule of Thumb**

To reinforce traversal driven modeling, you want your production edge labels to be in the direction that you will most commonly traverse and the materialized views to be in the less common direction.

### Loading data with DataStax Bulk Loader

There are no changes from Chapter 6 to the provided data or to how we load it for our example. However, recall the first five lines of `Sensor_send_Sensor.csv`, shown in Table 7-1.

*Table 7-1. The first five lines of data from the file `Sensor_send_Sensor.csv`*

| out_sensor_name | timestep | in_sensor_name |
|---|---|---|
| 103318117 | 1 | 126951211 |
| 1064041 | 2 | 1307588 |
| 1035508 | 2 | 1307588 |
| 1282094 | 1 | 1031441 |

In Chapter 6, our schema did not have a `timestep` on the `send` edge labels. Therefore, our loading process omitted the timestamps on the edge data.

However, our schema for this chapter uses `timestep` to cluster our edges. Therefore, when we load the exact same data with the same process, we will have edges with time on them. To see the code, please head to the data directory within this book's GitHub repository for the data and loading scripts for this chapter.

Let's apply our understanding of time, valid paths, and branching factor to refactor our queries from Chapter 6.

# Querying from Leaves to Roots in Production

We want to ask the same questions as before, but now we want to use time on the edges to consider only valid paths. Let's start with our first query and see when it communicated data to another sensor or tower.

## Where Has This Sensor Sent Information to, and at What Time?

This is the same question that we started with before, but we are using a different sensor this time: `104115939`. We want to add the `timestep` property into the map of results. This requires using the edge in our traversal and adding an additional element to our map. Let's look at the query in Example 7-1 and then at the example results. Then we will walk through the code below.

*Example 7-1.*

```
sensor = g.V().has("Sensor", "sensor_name", "104115939").next()
g.V(sensor).                              // look up the sensor
    outE("send").                         // walk out and stop on all edges
    project("Label", "Name", "Time").     // create a map for each edge
      by(__.inV().                        // traverse in
            label()).                     // values for the first key
      by(__.inV().                        // traverse in
           coalesce(values("tower_name"),// values for the 2nd key if a tower
                    values("sensor_name"))). // otherwise return sensor_name
      by(values("timestep"))              // values for the 3rd key: "Time"
```

And the results are:


```
{
  "Label": "Sensor",
  "Name": "104115918",
  "Time": "1"
},{
  "Label": "Sensor",
  "Name": "10330844",
  "Time": "0"
}
```


In Example 7-1, the query sets up as we have seen before. We create a traversal and populate the traversal pipeline on line 2 with a single vertex. On line 3, we move to all outgoing edges from the sensor. Line 4 uses `project` to create a map object with three keys: `Label`, `Name`, and `Time`. The values in the map for `Label` will be filled with the traversal from line 5: the label of the incoming vertex on the other side of the edge. The values in the map for `Name` will be filled with the try/catch pattern of the `coalesce` step on line 7: either the name of a tower or the name of a sensor. Last, the values in the map for the key `Time` will be filled with the traversal from line 10: accessing the property value `timestep` from the edge.

Let's use the pattern from Example 7-1 and follow any path to a tower, but we want to also look at the `timestep` values along the way.

## From This Sensor, Find All Trees up to a Tower by Time

The next query is the same one we set up in Chapter 6, but we are adding the `timestep` property from the edge into the result payload. From here, we will be able to understand which paths are valid and which are invalid. Let's look at the query in Example 7-2. We will delve into the details afterward.

*Example 7-2.*

```
sensor = g.V().has("Sensor", "sensor_name", "104115939").next()
g.V(sensor).                                 // look up a sensor
    as("start").                             // label it "startingSensor"
  until(hasLabel("Tower")).                  // until we reach a tower
  repeat(outE("send").                       // walk out and stop on the send edge
           as("send_edge").                  // label it "send_edge"
         inV().                              // walk into the adjacent vertex
           as("visited").                    // label it "visited"
          simplePath()).                     // remove cycles
  as("tower").                               // label it "tower"
  path().      // get path of vertices and edges from "start" to "tower"
    by(coalesce(values("tower_name",         // 1st object in the path is a vertex
                       "sensor_name"))).
    by(values("timestep"))                   // 2nd object in the path is an edge
```

Let's walk through the code from Example 7-2 before we show the results. Line 2 populates the traversal pipeline with a single vertex. The use of `until()/repeat()` on lines 4 and 5 uses the while/do pattern in Gremlin. Line 5 ensures each traverser in the pipeline accesses the `send` edge and labels it as `send_edge` so that we can reference it in the path object. Line 8 labels any vertex along the way as a visited vertex, while line 10 adds the label `tower` to the last vertex in the path. The last vertex on this walk will always be a tower due to the stopping condition in line 4.

The trickiest part of Example 7-2 occurs from lines 11 through 14. Here, we apply `by()` modulators in round-robin order to mutate the objects in the path structure so as to populate our query's results with meaningful information about each path.

Let's break this down.

Line 11 from Example 7-2 asks each traverser to populate its path object into the traversal pipeline. Every path will follow the structure `[Start, Edge, Vertex, … , Edge, Tower]`. This is true because we started at a sensor and then repeatedly accessed an edge and its adjacent vertex.

We use this pattern with the `by()` modulators on lines 12 and 14. The `by()` modulator on line 12 will map to the even-numbered objects in the path object `[0, 2, 4, … ]`. The objects at even-numbered positions in the path object are guaranteed to be vertices. For any vertex, we want to mutate the object in the path to include only the vertex's `tower_name` or its `sensor_name`; we use the try/catch pattern of the `coalesce()` step to do this.

On line 14, the `by()` modulator will map to the odd-numbered objects in the path object, `[1, 3, 5, … ]`. The odd-numbered objects in the path are guaranteed to be edges. We want the path object to show the `timestep` from a particular edge; we use `values("timestep")` to do this mutation.

In Example 7-3, we show two results from the query in Example 7-2. These results show the `labels` payload from the path object so that you can map each of the `as()` labels from the query to the path object. For space reasons, the results in Example 7-3 are the only time we will be showing the `labels` payload; this section of the results will be omitted throughout the rest of our examples.

*Example 7-3.*

```
...,
{
  "labels": [
    ["start"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited", "tower"],
  ],
  "objects": [
    "104115939",
    "0", "10330844",
    "1", "126951211",
    "2", "127620712",
    "3", "103318129",
    "4", "103318117",
    "5", "Bellevue"
  ]
},{
  "labels": [
    ["start"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited"],
    ["send_edge"], ["visited", "tower"],
  ],
  "objects": [
    "104115939",
    "0", "10330844",
    "1", "126951211",
    "2", "127620712",
    "3", "103318129",
    "0", "103318117",
    "5", "Bellevue"
  ]
}, ...
```

The first result in Example 7-3 is a valid path because time correctly follows in sequence: `0,1,2,3,4,5`. The second result is an invalid path because the sequence of time on the edges is out of order: `0,1,2,3,0,5`. The second result is an example of the communication path breaking after `timestep 3`.

The two paths shown in Example 7-3 point to the details of valid and invalid trees. The first path is valid because it follows time sequentially, whereas the second path does not. We have visualized the resulting paths in Figure 7-13 to see how one is valid and the other is invalid.

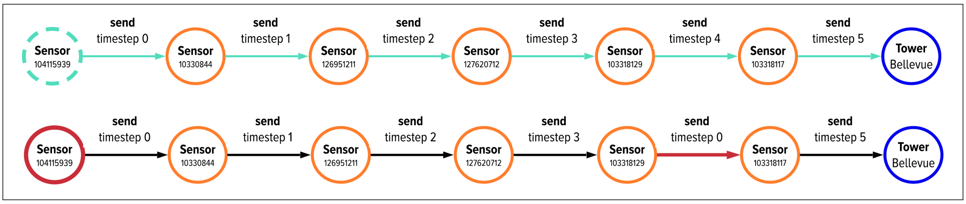



*Figure 7-13. Visualizing the results of Example 7-2 to see how one tree is valid and the other is invalid*

The top path in Figure 7-13 is valid because it follows an incremental pattern from start to finish. The bottom path in Figure 7-13 is broken because `Sensor 103318129` receives its data at `timestep 3`, but the next edge out of `103318129` occurs at an earlier time, `timestep 0`.

We need to consider only valid trees as we walk from a sensor up to a tower. Monitoring the value of `timestep` as we walk through our data is the final example for this section.

## From This Sensor, Find a Valid Tree

We want to use the pattern from Example 7-2, but we want to check the value on the `send` edges as we walk through the data. The idea is essentially to accomplish what you see in Example 7-4, but without hardcoding the `timestep` values.

*Example 7-4.*

```
sensor = g.V().has("Sensor", "sensor_name", "104115939").next()
g.V(sensor).                                   // look up a sensor
    outE("send").has("timestep", 0).inV(). // traverse edges with timestep = 0
    outE("send").has("timestep", 1).inV(). // traverse edges with timestep = 1
    outE("send").has("timestep", 2).inV(). // traverse edges with timestep = 2
    outE("send").has("timestep", 3).inV(). // traverse edges with timestep = 3
    outE("send").has("timestep", 4).inV(). // traverse edges with timestep = 4
    outE("send").has("timestep", 5).inV(). // traverse edges with timestep = 5
    path().                                // get the path from the sensor
      by(coalesce(values("tower_name",     // for the even position elements
                         "sensor_name"))). // get the vertex's ID
      by(values("timestep"))               // for the odd position elements
```

The query in Example 7-4 works if we already know how deep the tree is. For any sensor, we won't know this, and we'll need to use a counter variable. We will want to use a counter variable to start at 0 and increment by one until we find a tower.

Gremlin has a step for this: `loops()`. The `loops()` step keeps track of the number of times a repeat is executed; `loops()` starts at zero and will increment by one for every iteration of the repeat step.

*Loops()*
: The `loops()` step extracts the number of times the traverser has gone through the current loop.

We can use the counter from `loops()` and compare it to the value of an edge's `timestep`. Comparing the counter to an edge's `timestep` will give us the ability to consider only valid trees from our starting sensor to a tower.

Let's use `loops()` and create a filter on an edge. We want an edge to pass through the filter when its `timestep` is equal to the `loops()` variable. We want an edge to fail to pass through the filter if its `timestep` is not equal to the `loops()` variable. While this requirement seems rather contrived, it is very common to walk edges in a sequential fashion. The overarching problem and solution provide context and transferable solutions to a common application pattern.

Example 7-5 shows how to use `loops()` and create a filter on an edge in Gremlin.

*Example 7-5.*

```
sensor = g.V().has("Sensor", "sensor_name", "104115939").next()
g.V(sensor).as("start").          // look up a sensor, label it
   until(hasLabel("Tower")).      // until you reach a tower
   repeat(outE("send").           // traverse out to a send edge
            as("send_edge").      // label it "send_edge"
          where(eq("send_edge")). // filter: an equality test
            by(loops()).          // an edge passes if loops() is equal to
            by("timestep").       // the timestep on the edge
          inV().                  // walk to adjacent vertex
            as("visited")).       // label it "visited"
   as("tower").                   // guaranteed tower; label it "tower"
   path().                        // path from "start" to "tower"
    by(coalesce(values("tower_name",     // for the even position elements
                       "sensor_name"))). // get vertex's ID based on its label
    by(values("timestep"))               // for the odd position elements: time
```

And here are the results of Example 7-5; we omitted the `labels` payload from the `path()` object:


```
{ ... ,
  "objects": [
    "104115939",
    "0", "10330844",
    "1", "126951211",
    "2", "127620712",
    "3", "103318129",
    "4", "103318117",
    "5", "Bellevue"
  ]
}
```


Let's walk through the steps in Example 7-5. Line 2 fills the traversal pipeline with a starting vertex. Lines 3 through 9 set up our recursive walk from the sensor to any tower by accessing outgoing edges and then incoming vertices. Lines 6, 7, and 8 define a filter for an edge. A traverser passes through this filter if its `timestep` is equal to the loop counter. A traverser fails to pass through this filter if the edge's `timestep` is not equal to the loop counter.

The only traverser that will pass through this recursive loop and the filter will be the traverser that forms a valid walk from the starting sensor to the tower. We use the same pattern to format the path results and confirm that we found the only valid walk from sensor `104115939` up to the `Bellevue` tower.

## Advanced Gremlin: Understanding the where().by() Pattern

Using the `where().by()` pattern in Example 7-5 was probably a surprise to you.

We would like to show you a common way people try to solve this problem and then explain why it doesn't work, to help you understand a deeper topic from the Gremlin query language.

### Understanding a common Gremlin mistake: Overloading has()

Most people would start by using `has("timestep", loops())` as a filter on the edges. We will take a look at using it in Example 7-6 and then we will explain why it is wrong.

> The query in Example 7-6 doesn't accurately answer the question for this chapter. It is included for educational purposes.

*Example 7-6.*

```
g.V(sensor).
    until(hasLabel("Tower")).
    repeat(outE("send").as("send_edge").
           has("timestep", loops()). // this does not work; details in text
           inV().as("visited")).
    as("tower").
    path().
      by(coalesce(values("tower_name", "sensor_name"))).
      by(values("timestep"))
```

The results of Example 7-6 follow; we omitted the `labels` payload from the `path()` object:

```
{ ... ,
  "objects": [
    "104115939",
    "0", "10330844",
    "1", "126951211",
    "2", "127620712",
    "3", "103318129",
    "4", "103318117",
    "5", "Bellevue"
  ], ... ,
  "objects": [
    "104115939",
    "0", "10330844",
    "1", "126951211",
    "2", "127620712",
    "3", "103318129",
    "0", "103318117", //incorrect result: time is out of order: 3, 0, 5
    "5", "Bellevue"
  ]
}, ...
```

The results for Example 7-6 exactly match the results from Example 7-2. This is because the use of `has("timestep", loops())` is overloaded, and every traverser passes for all edges.

The mistake we are making here is that we are asking the question "Is `loops()` accessible," instead of "Does the value of `loops()` match the value of the `timestep` property on the edge?"

Let's dig in and see why.

The use of the `has()` step in Example 7-6 creates a filter with the structure `has(key, traversal)`. With this structure, the `has()` step creates a traversal that starts from the property value `timestep`. The edge will pass through the `has()` filter if the traverser

survives. The condition that determines whether a traverser survives is `loops()`, which will always work because `loops()` will return a value.

Essentially, we created the logic of `has(True)` in Example 7-6.

The overloaded use of `has(key, traversal)` is one of the most common mistakes we find when helping Gremlin users write recursive queries. We hope this helps you avoid making that same mistake.

### Resolution: The where().by() pattern

If `has("timestep", loops())` doesn't work, why does the `where().by()` pattern work?

Let's dig into why.

In Example 7-5, we used the following Gremlin pattern to create an edge filter:

```
where(eq("sendEdge")).
        by(loops()).
        by("timestep")
```

The basic form of `where()` in Gremlin is `where(a, pred(b))`. Our usage applies the shorthand of `where(pred(b))`, in which the incoming traverser is implicitly assigned to `a`.

Since the incoming traverser was labeled `sendEdge`, you actually have:

```
where("sendEdge", eq("sendEdge"))
```

This pattern will only ever evaluate false if you use two different `by()` modulators, which are then applied to `sendEdge` and `eq("sendEdge")`, respectively—or in this case, when the `by()` modulators emit two different values from the same edge.

Our two `by()` modulators are emitting the values for `loops()` and `timestep`, respectively. If those values are different, the expression evaluates to false and the incoming traverser is eliminated.

At this point, we have completed exploring all concepts required for walking from sensors up to towers. Last up for this example: we go back to the top of our trees and walk from the towers down to the sensors.

## Querying from Roots to Leaves in Production

The final technical section of this chapter uses the sensor network data to avoid the branching factor issues as we walk from towers to sensors. The queries here apply the sorted order of `send` edges to navigate specific edges and solve the error we concluded with in Chapter 6.

Let's start with the tower we explored in the last chapter to answer our first query.

## Which Sensors Have Connected to Georgetown Directly, by Time?

For this question, we want to inspect the `Georgetown` tower and see how many messages it received and at what time it received each message. As always, we want to construct a JSON object that shows which sensor sent it and at what time. Let's look at the query in Example 7-7 and then at some results.

*Example 7-7.*

```
1 tower = dev.V().has("Tower", "tower_name", "Georgetown").next()
2 g.V(tower).
3     inE("send").
4     project("Label", "Name", "Time"). // create a map for each edge
5       by(outV().label()).             // value for the first key "Label"
6       by(outV().                      // value for the second key "Name"
7          coalesce(values("tower_name"),    // if a tower, return tower_name
8                   values("sensor_name"))). // else, return sensor_name
9       by(values("timestep"))         // value for the third key "Time"
```

And here are the results of Example 7-7:

```
{
  "Label": "Sensor",
  "Name": "1302832",
  "Time": "3"
},{
  "Label": "Sensor",
  "Name": "1002688",
  "Time": "2"
},...,{
  "Label": "Sensor",
  "Name": "1306931",
  "Time": "1"
}
```

This example follows the same construction pattern with `project()` that we have been using for most of our queries. Let's walk through what this query is doing, one line at a time.

On line 2 of Example 7-7, we populate our traversal with one vertex: the `Georgetown` tower. Line 3 splits the one traverser into many traversers; one traverser for each of the seven adjacent edges. This means that the `Georgetown` tower has a branching factor of 7, and we now have seven traversers to process in our pipeline. Lines 4 through 9 tell each traverser how to report back the necessary data into the result payload. We create a map with the keys `Label`, `Name`, and `Time` on line 4. Line 5 fills the key `Label` with the label of the outgoing vertex. Lines 6 through 8 fill the `Name` key with the par-

tition key from the outgoing vertex. Last, line 9 fills the `Time` key with the edge's `timestep`.

We have used this pattern multiple times to construct JSON payloads of our graph data. Hopefully this is becoming ingrained as a useful Gremlin step for shaping query results.

Believe it or not, we have only one more question to ask for this chapter. We want to walk from the `Georgetown` tower to find valid paths down to sensors.

## What Valid Paths Can We Find from Georgetown Down to All Sensors?

For this query, we have to define where we want to start in time. Our examination of the results of Example 7-7 shows that we can find trees that end at `timestep` `3`, `2`, or `1`. Let's look at trees that ended at `timestep 3`.

For this query, we are first going to sketch out our approach in pseudocode, as shown in Example 7-8.

*Example 7-8.*

```
Question: What Valid Paths Can We Find from Georgetown Down to All Sensors?
Process:
   Initialize a counter variable
   For a total of counter + 1 times (to account for the zero-th edge),
   Do the following:
       Walk to incoming send edges
       Create a filter to compare an edge's timestep with the counter
       Decrease the counter by 1
   Show and shape the path from the tower to the ending sensor
```

To write this type of query, we need to dive into a new Gremlin concept: the `sack()` operator.

As we walk from a tower down through different levels of the tree, we want a data structure that tracks how many steps we have taken. In Example 7-5, we used the `loops()` step. `Loops()` increments by one, but we need to decrease by one for each depth.

We need something different.

We can customize a variable in a Gremlin traversal with the `sack()` step. You can think of the sack step as giving each traverser a backpack at the beginning of its journey in your graph data. You can initialize the sack with whatever you would like. As your traverser moves through graph data, it can mutate the contents of its sack according to what it is processing from the graph data.

`Sack()`
: A traverser can contain a local data structure called a sack. The `sack()`-step is used to read and write to a traverser's sack.

`WithSack()`
: The `withSack()` step is used to initialize the sack data structure.

In the next query, we will start at `timestep 3` and walk through edges with `timestep` values of `2`, `1`, and `0`, respectively. You can change this to any start time for additional practice. We picked `start = 3` to teach the concepts in this query.

Let's see how to use `repeat()` with `times()` and the `sack()` operator in Gremlin to answer the pseudocode we outlined in Example 7-8. The query is in Example 7-9.

*Example 7-9.*

```
1 start = 3
2 tower = dev.V().has("Tower", "tower_name", "Georgetown").next()
3 g.withSack(start).  // every traverser starts with a sack with a value of 3
4   V(tower).as("start").           // look up Georgetown
5   repeat(inE("send").as("send_edge").// traverse to incoming edges
6          where(eq("send_edge")).     // create an equality filter:
7            by(sack()).               // test if the sack() value
8            by("timestep").           // equals the edge's timestep
9          sack(minus).                // decrease the sack's value
10           by(constant(1)).          // by 1
11         outV().as("visited")).      // traverse to adjacent vertex
12   times(start+1).                   // do lines 5-10 four times
13   as("tower").          // this vertex passed all edge filters
14   path().               // get the path to it starting from Georgetown
15     by(coalesce(values("tower_name", // first object in path is a vertex
16                        "sensor_name"))).
17     by(values("timestep"))            // second object in path is an edge
```

Let's step through the query one line at a time and then take a look at the results.

Line 1 of Example 7-9 initialized a starting variable to be 3. We will use this variable multiple times in the query. The first place we use the variable is on line 3, where we initialize a traverser's sack to be 3. Line 4 populates our traversal pipeline with the `Georgetown` tower. Then we see the `repeat()/times()` pattern on lines 5 and 12. Here, we use the value `start + 1` as the stopping condition for any traverser. This means that the traversal from lines 5 through 12 will be completed after `start + 1 =` 4 iterations.

Within the repeat clause, we construct a filter for every edge that we process. We use the same `where()/by()` pattern that we did before. This time, however, we replace `loops()` with `sack()`, which means that an edge's `timestep` will be compared to the value in `sack()`.

Let's walk through how `sack()` works within this loop.

The first time we process the traversal within the repeat step, each traverser will have 3 stored within its sack. This means that the first time we use the filter on lines 6, 7, and 8, we will compare an edge's `timestep` to the integer 3. Only the edges adjacent to `Georgetown` with a `timestep` of 3 will pass through this filter.

On line 9, we mutate the value in a traverser's sack. We decrease the sack's value with the `sack(minus)` step. The `by()` modulator on line 10 tells the traverser how much to subtract from the sack. We want to subtract one, so we use `by(constant(1))`.

On line 11, we move to the other vertex, and line 12 checks the looping condition. Lines 14 through 17 format the path results, as we have done many times. The results of Example 7-9 follow; we omitted the `labels` payload from the `path()` object:

```
{...,
  "objects": [
    "Georgetown",
    "3", "1302832",
    "2", "1059089",
    "1", "1255230",
    "0", "1248210"
  ] , ... ,
  "objects": [
    "Georgetown",
    "3", "1302832",
    "2", "1059089",
    "1", "1302832",  // cycle
    "0", "1010055"
  ]
}, ...
```

A keen observer will see an unexpected result. The second object contains a repeated sensor, `1302832`, even though the path follows the correct time values. We need to remove cycles from our results, as we did in Chapter 6.

The resulting query, shown in Example 7-10, is the same as before, but with this new step on line 12.

*Example 7-10.*

```
1 start = 3
2 tower = dev.V().has("Tower", "tower_name", "Georgetown").next()
3 g.withSack(start).
4   V(tower).as("start").
5   repeat(inE("send").as("send_edge").
6          where(eq("send_edge")).
7            by(sack()).
8            by("timestep").
9          sack(minus).
```

```
        by(constant(1)).
      outV().as("visited").
      simplePath()).       // remove cycles
times(start+1).
as("tower").
path().
  by(coalesce(values("tower_name",
                     "sensor_name"))).
  by(values("timestep"))
```

The results of Example 7-10 follow; we omitted the `labels` payload from the `path()` object:


```
{...,
  "objects": [
    "Georgetown",
    "3", "1302832",
    "2", "1059089",
    "1", "1255230",
    "0", "1248210"
  ]
}, ... ,
  "objects": [
    "Georgetown",
    "3", "1302832",
    "2", "1059089",
    "1", "1255230",
    "0", "1280634"
  ]
}
```


Inspecting the result payloads, we see two different valid trees that start from the `Georgetown` tower. One tree ends at sensor `1248210`. The other ends at sensor `1280634`.

And that is it for our query creation!

We have successfully addressed the errors from the end of Chapter 6 and are able to walk to and from the leaves and roots in our example data.

# Applying Your Queries to Tower Failure Scenarios

As a data engineer for Edge Energy, your final task is to apply what you have built to address Edge Energy's larger problem: what is the impact of a shutdown or tower failure on the network?

The art of understanding your data and graph technology derives from integrating multiple components to solve complex problems. Over the past two chapters, we have been setting up data, schema, and queries to do just that: use the relationships within our data to provide insights into a network's dynamic and evolving topology.

So how do we integrate our results over the past two chapters to resolve Edge Energy's complex problem? We break down the company's complex problem using the tools we have set up.

We have been querying around the `Georgetown` tower for a while now. Let's revisit the image we saw in Chapter 6 and think about the impact if the `Georgetown` tower were to fail. The image in Figure 7-14 shows the `Georgetown` tower in orange. The green arteries are all nearby sensors. The blue diamonds are other nearby towers.

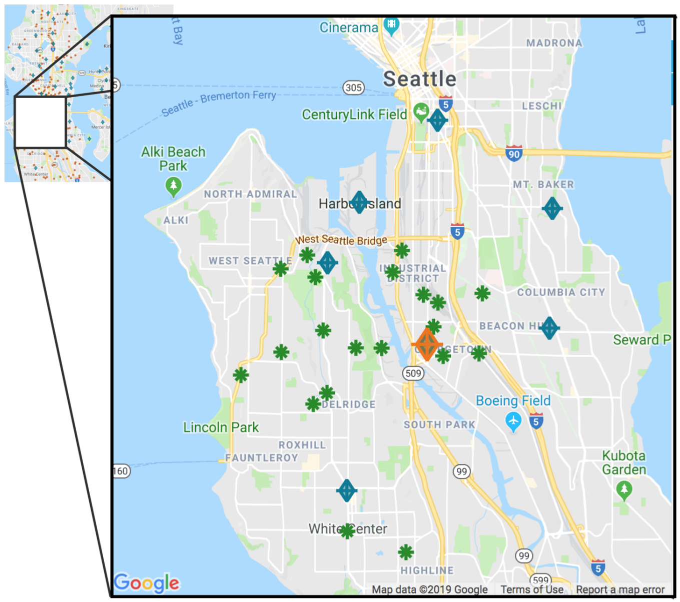



*Figure 7-14. Visualizing the sensors and towers around Georgetown; network edges not shown for image clarity*

Consider what would happen if the `Georgetown` tower were to go down. Which sensors, if any, will we lose connection to? Will they be only the sensors that surround the tower?

Let's query our graph and let the data tell us what would happen. We have ironed out two tools to use to answer this question:

1. We can report, for any tower, all of the sensors that communicated with it.
2. For any sensor, we can tell which towers connected with.

To resolve Edge Energy's complex network failure problem, we can apply the following procedure for the `Georgetown` tower:

1. Get a list of sensors that connected with `Georgetown` in any time window.
2. For each sensor, query the network to see if they used a different tower in that time window.

Let's answer each question using the queries we already built.

### Get a list of sensors that connected with Georgetown in any time window

Example 7-11 shows what we did in the accompanying Studio Notebook:

*Example 7-11.*

```
Question: Get a list of sensors that connected with Georgetown in any time window
Process:
    Wrap our query from a tower to sensors in a method: getSensorsFromTower()
    For each step in time:
        Find all sensors that connected with Georgetown
    Create a unique list of the sensors
```

The code for the pseudocode in Example 7-11 is shown in Example 7-12.

*Example 7-12.*

```
// wrap our query of valid paths in a method called getSensorsFromTower
def getSensorsFromTower(g, start, tower){
    sensors = g.withSack(start).V(tower).
                        repeat(inE("send").as("sendEdge").
                               where(eq("sendEdge")).
                                   by(sack()).
                                   by("timestep").
                               sack(minus).
                               by(constant(1)).
                               outV().
                               simplePath()).
                        times(start+1).
                        values("sensor_name").
                        toList()
    return sensors;
}
atRiskSensors = [] as Set;          // create a list of sensors

tower = g.V().has("Tower", "tower_name", "Georgetown").next();
for(time = 0; time < 6; time++){  // loop through a window of time
    // all sensors into Georgetown's list at this time via getSensorsFromTower()
    atRiskSensors.addAll(getSensorsFromTower(g, time, tower));
}
```

The main result from Example 7-12 is the object `atRiskSensors`. This is a list of all sensors that had valid communication paths with the `Georgetown` tower. The first four sensors are:

```
"1302832",
"1059089",
"1290383",
"1201412",
...
```

There is one last thing we need to know to provide proactive information to Edge Energy. We need to know which of the other towers the at-risk sensors communicated with.

#### For each at-risk sensor, find all towers it communicated with

Example 7-13 shows what we did in the accompanying Studio Notebook.

*Example 7-13.*

```
Question: For each at-risk sensor, find all towers it communicated with
Process:
    Wrap our query from a sensor to towers in a method: getTowersFromSensor()
    For each sensor in atRiskSensors:
        For each step in time:
            Find the towers the sensors connected with
        Add to a map of the unique towers a sensor connected to
    Find sensors that connected only to Georgetown
```

As we analyze all of the paths in our data, we ultimately are looking for sensors that uniquely connected to `Georgetown`. The code for our pseudocode in Example 7-13 is shown in Example 7-14.

*Example 7-14.*

```
// wrap our query of valid paths in a method called getTowersFromSensor
def getTowersFromSensor(g, start, sensor) {
    towers = g.withSack(start).V(sensor).
                until(hasLabel("Tower")).
                repeat(outE("send").as("sendEdge").
                       where(eq("sendEdge")).
                         by(sack()).
                         by("timestep").
                       inV().
                       sack(sum).
                       by(constant(1))).
                values("tower_name").
                dedup().
                toList()
```

```
    return towers;
}

otherTowers = [:];                                 // create a map

for(i=0; i < atRiskSensors.size(); i++){          // loop through all sensors
    otherTowers[atRiskSensors[i]] = [] as Set;    // initialize the map for a sensor
    sensor = g.V().has("Sensor", "sensor_name", atRiskSensors[i]).next();
    for(time = 0; time < 6; time++){       // loop through a window of time
        // use getTowersFromSensor to add all towers
        // into the map for this sensor at this time
        otherTowers[atRiskSensors[i]].addAll(getTowersFromSensor(g, time, sensor));
    }
}
```

The main result from Example 7-12 is the object `otherTowers`. This is a `hashMap` of all unique towers that had valid communication paths from the starting sensor. Let's take a look at the first few entries in `otherTowers`.

*Example 7-15.*

```
{   "1035508": ["Georgetown", "WhiteCenter", "RainierValley"]
},{ "1201412": ["Georgetown", "Youngstown"]
},{ "1255230": ["Georgetown"]
}, ...
```

Example 7-15 brings everything from the past two chapters together into one payload. We interpret this data to mean that `1035508` has two other options in the event that `Georgetown` fails: `WhiteCenter` or `RainierValley`. However, for the time window we looked at, `1255230` is a sensor at risk because it communicated only with `Georgetown` during the time window we studied.

We visualize all at-risk sensors from Example 7-15 in Figure 7-15.

The map in Figure 7-15 visualizes a network failure scenario for the `Georgetown` tower. The `Georgetown` tower is shown in red. All sensors that communicated *only* with `Georgetown` during a particular time window of interest are shown in orange. All sensors that successfully communicated with other nearby towers are shown in green. The other towers are shown as blue diamonds.

Let's step back a bit to understand where we are.

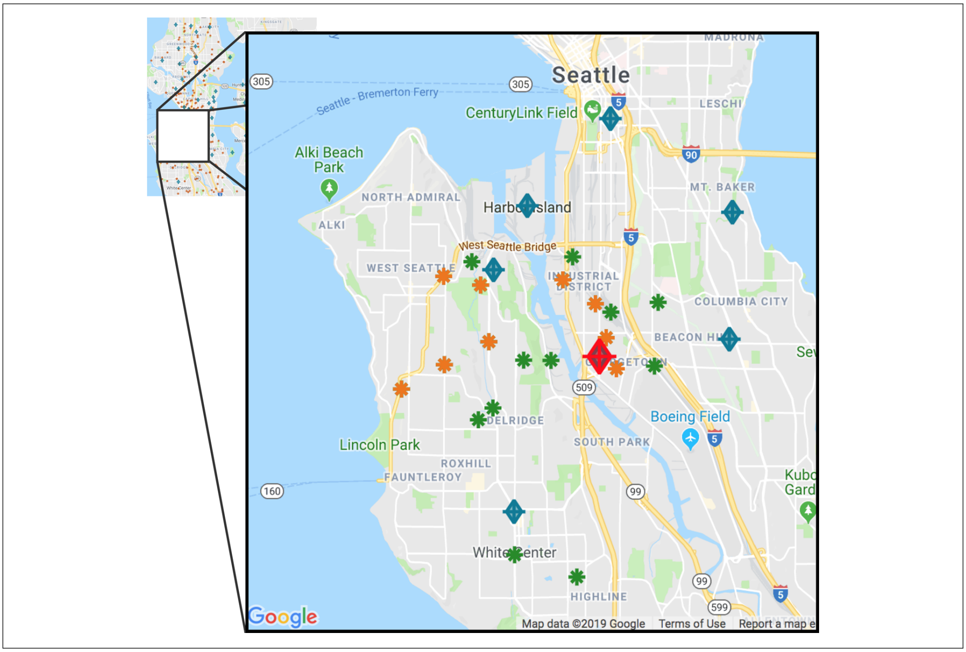



*Figure 7-15. Simulating failure and visualizing the at-risk sensors around the `George town tower`; network edges are not shown for image clarity*

## Applying the Final Results of Our Complex Problem

What we have built toward is the start of a proactive conversation with the Edge Energy team. We can take these results, the data, and its observable relationships within the network to determine Edge Energy's next step.

The orange sensors are not failures to report back to Edge Energy. They represent sensors that are at risk. Examining the geo-location from Figure 7-13 reveals that there are many nearby sensors and towers that each at-risk sensor can connect with. Only through additional observations over time will Edge Energy be able to fully understand any sensor's individual risk in the network.

With distributed graph technology, we are helping Edge Energy monitor its network. It can use the evolving structure of this graph's topology to be proactive about different network failure scenarios.

# Seeing the Forest for the Trees

Our work over Chapters 6 and 7 explored the hierarchical structure of time series data from a self-organizing network of sensors so that we could solve a complex problem about Edge Energy's dynamic network. We stitched together our queries and

understanding of the data to help Edge Energy with a complex problem: how to use time series data in a graph to be proactive about network failures.

Who knew that traversing through trees could be a walk in a park?

If you haven't already done so, we recommend you take all of this for a drive yourself. The accompanying Studio Notebooks, found at *https://oreil.ly/graph-book* walk through each of these queries, with a few more bonus items not mentioned in these chapters.

So far in this book, we have covered the data models and queries for two of the most popular graph models in distributed systems: neighborhoods and hierarchies. The next chapter introduces another popular data pattern. We will be introducing and using the third most popular data model and queries for distributed graph applications: network paths.

## CHAPTER 8

# Finding Paths in Development

Pathfinding in graph data is the next most popular use of graph technology, after neighborhood retrieval and unbounded hierarchies.

In addition to interviewing graph users around the world for this book, we also spent a significant amount of time working with them. More often than not, our working sessions centered on finding unknown paths within graph data.

During one of those working sessions, we were training a team on popular pathfinding techniques. We were using a graph of flight paths between airports to reason about flight patterns between cities.[1] We started our exercise with the two most popular questions about air travel: how many direct connections are there from this specific airport? And how many airports are reachable within two flights?

The troubleshooting discussion during the workshop led me to question how people use path information to make an informed decision.

One particularly interesting implication is related to trust.

How do you decide if you trust somebody? You trust your friends. And you probably trust friends of your friends more than you trust a random stranger. Why is that?

It is your trust in different paths between you and something else that motivates and informs your preferences.

1 Kelvin Lawrence, *Practical Gremlin: An Apache TinkerPop Tutorial*, January 6, 2020, *https://github.com/krlawrence/graph*.

## Chapter Preview: Quantifying Trust in Networks

There are four main sections of this chapter.

We'll first cover some more examples of how we all use paths to quantify trust. Then we'll start with an overview of the required concepts from mathematics and computer science for working with paths in graph data. After that, we'll set up this chapter's example, in which we'll be working with, querying, and finding paths throughout the Bitcoin trust network to answer the fundamental question: how much should you trust someone before you interact with them? The final section of this chapter applies path queries to the Bitcoin trust network. We will start with exploring and understanding trust within the data. Then we'll show you how to use path queries to inform a decision about whether to trust a particular Bitcoin wallet.

We'll conclude the chapter with a mathematical quantification of trust that leads to a problem we'll solve in the next chapter.

## Thinking About Trust: Three Examples

The theme of using data to quantify trust extends beyond the air travel example previously mentioned. The correlation between trust and paths in graph data applies to almost all of the path applications we work on with our customers around the world.

We have seen this in how people use social media, in how detectives build criminal cases, and in logistics optimizations.

### How Much Do You Trust That Open Invitation?

Think about the social media platform that you use most regularly.

How do you determine whether you are going to accept that connection, follower, or friend request?

If you are like most people, you undertake a very common process for new connection requests. Typically, you first look at the shared connections between you and the potential new friend, connection, or follower. Figure 8-1 offers a graph of the possible connections you might look for.

You likely ask yourself, "How many friends do I have in common with the person?" Is it 3 shared friends or 30? Then you look at the quality of those shared connections. Are any of your closest friends or family members in the list of shared connections? Are your shared connections all from a specific point in your life, like a particular job or school?

Your analysis consists of walking through the quantity and quality of your shared connections. You are using the paths between you and the new connection to contextualize and inform how you know that person. Ultimately, it is your trust in those paths that leads you to accept or reject that new connection.

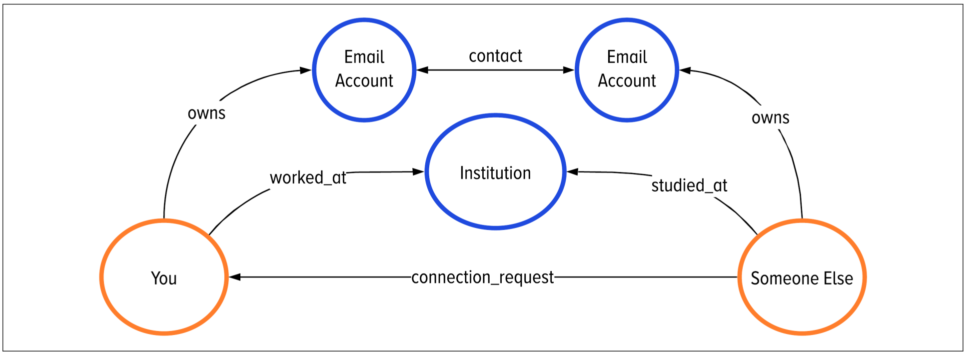



*Figure 8-1. An example of seeing paths between you and an open invitation on social media*

Accepting a new connection on social media starts with your shortest path and then naturally evolves into the quality and context of those paths.

Social media helps us quantify how much we trust anyone new. We use our shared connections to construct a story about how we know someone and therefore whether we trust them.

It may be something that we now do naturally every time we engage with our networks. But this isn't the first use of this technique. Investigators have been using trusted sources to create connections between two previously disconnected individuals for a long time.

## How Defensible Is an Investigator's Story?

The long history of criminal investigations, together with rising volumes of data and emerging patterns of graph technology, serves as the perfect environment for quantifying trust in relationships across data.

A detective's work is to pull together sources of information to understand how two individuals are connected. Detectives obtain access to records by subpoenaing data sources related to the case. Then investigators unify the data sources and directly search for unknown connections within their open case. Figure 8-2 shows a graph of some of these data sources. The figure depicts what we came up with for a detective's story, but you should think about this conceptually, too.

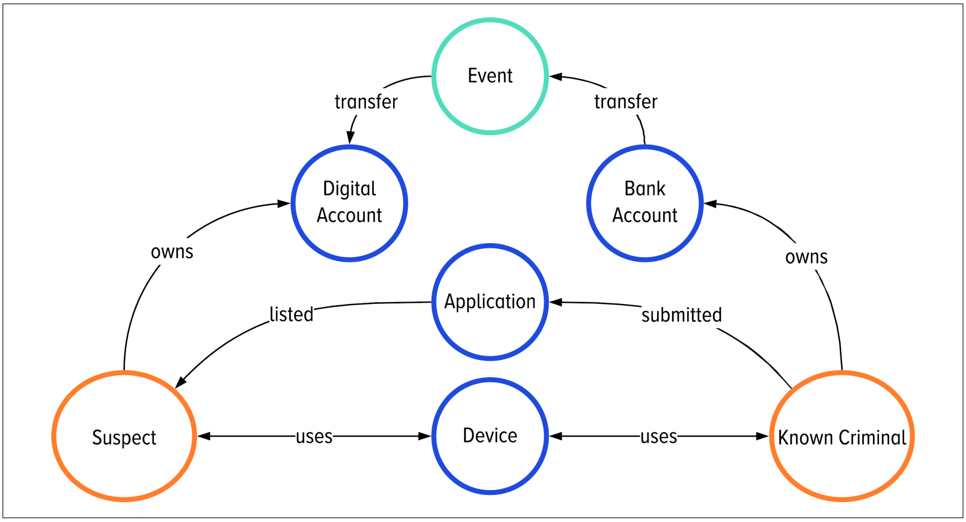



*Figure 8-2. An example of analyzing paths to inform an investigation into a suspicious person*

Drawing correlations about the connections between two individuals in a criminal investigation uses paths through data to tell stories about what happened. The investigators are reporting information governed by law; they have to trust the quality of the connections that construct the story.

On a less serious scale, you do the same type of investigations when you make a decision regarding your personal flight schedule. You make decisions about your air travel based on the context and quality of the route you purchase in the same way investigators derive conclusions about a case.

Let's look at a third example of using paths in networks to quantify trust.

## How Do Companies Model Package Delivery?

A logistics company might seek to minimize costs and time along its delivery routes. As part of that minimization, it may consider the number of times a package has to be transferred between the warehouse and your front door. Fewer transfers minimizes the number of chances for a package to be lost or misplaced. We drew a graph that represents this network in Figure 8-3.

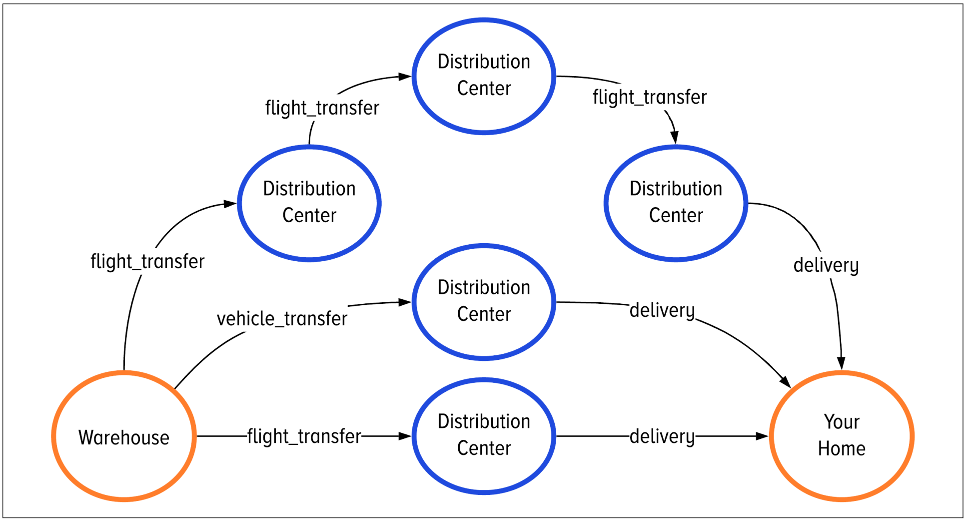



*Figure 8-3. An example of analyzing paths to determine an optimal route for a logistics company*

Figure 8-3 depicts how a package travels from a warehouse to your home. You see three potential paths, each with different combinations of length and types of transfer. Depending on a multitude of factors, one path in our logistics network may be more trusted than another.

For example, if you are someone who watches your package's path, you also feel the effect of route optimization for shipping. The more stops you see your package take, the lower your trust in its on-time arrival.

Route optimization is one of the most popular uses of graphs in computer science. Whether you are making decisions for personal travel or waiting for a package, the most trusted solutions seek the shortest path through the data.

It is the trust in the path between the source and the destination that matters.

Quantifying trust between two concepts through understanding shared relationships is (probably) the most relatable and approachable application of distributed graph technology today.

# Fundamental Concepts About Paths

Pathfinding queries are popular uses of graph technology when you do not know exactly how to walk between vertices in your graph.

However, discovering paths throughout graph structure may become a double-edged sword: on the one hand, pathfinding with graph technology will provide you with

short and elegant solutions; on the other hand, naive pathfinding queries may quickly get out of control.

Pathfinding questions are simple to ask but expensive to compute. This is where things can get out of hand very quickly.

Let's start by walking through the fundamental problem definitions for discovering paths in graph structure.

## Shortest Paths

In this chapter, we will introduce shortest paths according to a path's distance.

Recall from Chapter 2 the definition of distance as the smallest number of edges it takes to walk from one vertex to another. The *shortest path problem* is to find the path with the smallest distance, or shortest walk, from one vertex to another in your graph. Here are the four terms and their definitions that we will be applying throughout the next two chapters.

*Path*
: A path in a graph is a sequence of consecutive edges in a graph.

*Length*
: The length of a path is the number of edges in the path.

*Shortest path*
: The shortest path between two vertices is the path that connects the two vertices and has the shortest length or distance.

*Distance*
: The distance between two vertices in a graph is the number of edges in a shortest path.

In Figure 8-4, there are three ways to walk from A to D:

1. A → D
2. A → C → D
3. A → B → C → D

The shortest path is the path with the smallest distance. That is the path from A to D, which has a distance of 1. The other paths have distances of 2 and 3, respectively.

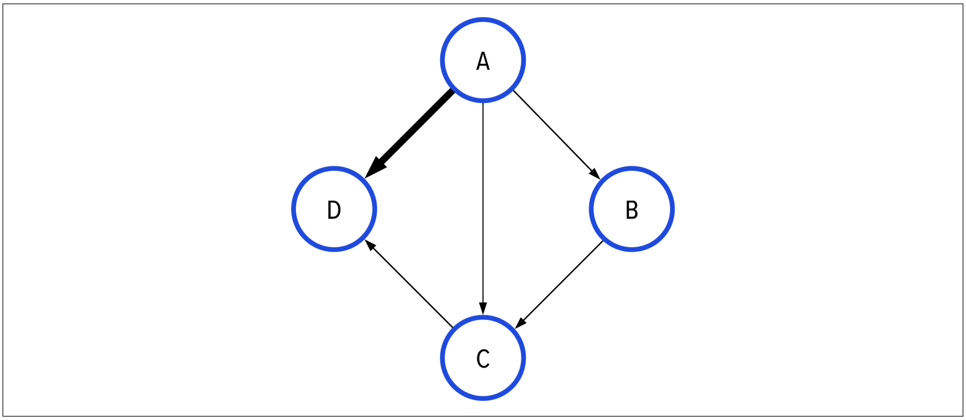


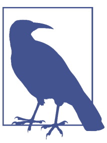

*Figure 8-4. An example of the shortest path from A to D*

There are three types of shortest path problems:

*Shortest path*
: The goal of a shortest path problem is to discover the smallest distance walk from A to B.

*Single-source shortest path*
: The goal of a single-source shortest path problem is to discover the smallest-distance walk from A to all other vertices in the graph.

*All-pairs shortest path*
: The goal of an all-pairs shortest path problem is to discover the smallest-distance walk between any two vertices in the graph.



> These definitions give us a classification of the three types of path-finding problems that you may have run into or will run into. This chapter focuses on solutions to the first type of problem: finding the shortest path between two known points.

Any solution to a path problem relies on understanding how to procedurally walk through graph data. Let's dig into depth-first search (DFS) and breadth-first search (BFS), two fundamental techniques for finding shortest paths.

*Depth-first search*
: Depth-first search is an algorithm for traversing graph data structures. It explores a path as deep as possible along each branch before backtracking.

*Breadth-First Search*

Breadth-first search is an algorithm for traversing graph data structures. It explores all of the neighbor vertices at the present depth prior to moving on to the vertices at the next depth level.

You may be wondering: "Why do we need to go into DFS versus BFS?"

First, most engineers start their research about pathfinding by searching for information on a certain pathfinding algorithm. To us, that is a backwards approach. Second, because paths are so natural to understand, it is easy to confuse the solution with the underlying problem.

You first need to understand which path problem you are trying to solve before you apply a certain algorithm.

## Depth-First Search and Breadth-First Search

Depth-first search and breadth-first search are two of the most popular ways to illustrate procedural visitation of graph-structured data. Diving deep into understanding each technique gives you the foundation you need to explore the world of pathfinding algorithms, because at some level, all other solutions to pathfinding problems build upon these two techniques.

The difference between the two approaches is easy to understand. Depth-first search prioritizes exploring one path as deeply as you can before returning to a different path. Breadth-first search prioritizes exploring all paths up to a certain distance before moving deeper into the data.

Let's take a look at these differences in Figure 8-5. As you walk through the figure, the main idea is to consider the order in which a vertex is visited for each process; we call this the *visited set*. We numerically label each vertex in Figure 8-5 with the order in which it is visited (or reached) by each algorithm.

For each graph in Figure 8-5, the goal is to walk procedurally from the starting vertex at the top to the end. The graph on the left shows the order in which every vertex is visited according to DFS. Here, you see that each branch is explored until its end before you return back to the top to select a different path. The graph on the right shows the order in which every vertex is visited according to BFS. Here, you see that each level or neighborhood is fully explored before you move deeper into the graph.

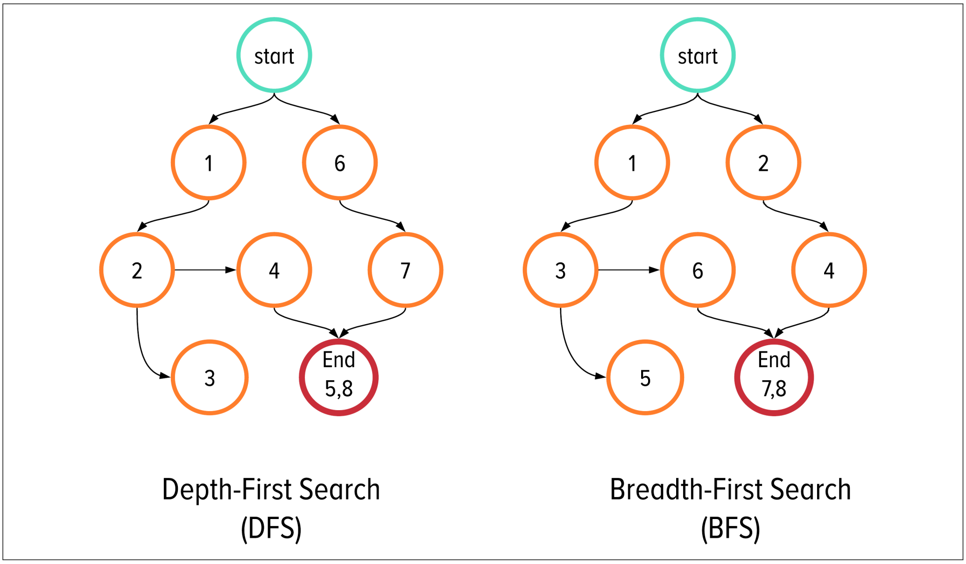



*Figure 8-5. Illustrating the visited set for the two most common graph-searching algorithms*

The implementation details between DFS and BFS come down to which data structure you use. DFS uses a last in, first out (LIFO) stack. You can remember this by visualizing a stack. Stacks are typically thought of as vertical structures, just like how DFS explores data deeply before going wide. BFS uses a first in, first out (FIFO) queue. You can remember this by visualizing a queue. Queues are typically thought of as horizontal structures, just like how BFS explores widely before going deep.

However long it takes you to think like a graph, it is vital that you understand the runtime and overhead required to process your data. So keep practicing how to procedurally think through how much data you need to visit during a traversal. From there, you can quantify an expectation for how long a traversal or algorithm will run or the overhead it requires for your data.

## Learning to See Application Features as Different Path Problems

Think about the last time you used LinkedIn. You likely opened the LinkedIn application to search for someone else. When you found candidates, you received a metric indicating how closely connected you were to each person in your search results. You knew right away whether someone was a first-, second-, or third-degree connection.

Now think like an engineer working for LinkedIn.

In this scenario, you are designing the connected badge feature that you just used. It is a requirement that any user of LinkedIn knows the distance from themselves to

anyone else when they search. From there, you and your engineering team have a long list of approaches to consider.

Do you precalculate all distance values for the connected badge by solving the all-pairs shortest path problem for your graph? If you do, what happens when new connections are added to or removed from LinkedIn's network?

What are the end user's expectations for knowing their connectedness to another person? What requirements can you relax in order to prioritize speed of delivering the information to the end user?

Though presented in the context of pathfinding at LinkedIn, all of these questions are common considerations for any team wanting to use path distance in an application.

To answer any of those questions about your application's design, you need to understand the performance implications of processing graph-structured data. And the fundamental approach to walking through graph-structured data to solve problems at LinkedIn scale builds procedures off of BFS or DFS.

We will be using these fundamental techniques in the coming sections as we explore the example data and find paths of trust throughout it. To that end, let's introduce the data for this chapter's example and apply shortest paths to our sample problem.

# Finding Paths in a Trust Network

Distance between concepts quantifies trust.

To bring that axiom to life, our running example from now until the end of Chapter 9 dives into the world of Bitcoin. Exploring a network of trust between Bitcoin traders creates an interesting intersection between paths in graph data and trust. Ironically, the advent of Bitcoin centers on a distrust of centralized institutions.

In this section, we will introduce the data, walk through a brief primer on Bitcoin terminology, and develop our data model.

## Source Data

We will be exploring a network of people who trade Bitcoin on the Bitcoin OTC (Over The Counter) Marketplace. The Bitcoin OTC Marketplace allows its members to rate how much they trust other members, and those ratings form who-trusts-whom networks, which we will be using in the dataset. These ratings are given on a scale of [–10, 10]. You will see the ratings in the details to come, but we won't use

them in our queries until Chapter 9. The data comes from the research work of Srijan Kumar et al. and can be found on the Stanford Network Analysis Platform.[2] [3]

> Stanford Network Analysis Platform (SNAP) is a general-purpose network analysis and graph mining library.

Each line in the dataset has one rating, sorted by time, with the following format:

```
SOURCE, TARGET, RATING, TIME
```

The meaning for each piece of the data is as follows:

*Source*
: The vertex ID of the rater

*Target*
: The vertex ID of the ratee

*Rating*
: The source's rating for the target, ranging from –10 to +10 in steps of 1

*Time*
: The time of the rating, measured as seconds since epoch

Let's look at the first five lines of the data in Example 8-1.

*Example 8-1.*

```
$ head -5 soc-sign-Bitcoinotc.csv
6,2,4,1289241911.72836
6,5,2,1289241941.53378
1,15,1,1289243140.39049
4,3,7,1289245277.36975
13,16,8,1289254254.44746
```

Let's examine the first line of data from Example 8-1: `6,2,4,1289241911.72836`. This means that the person with ID `6` trusts the person with ID `2` a total of `4`. This rating

2 Kumar, Srijan, et al. "Edge Weight Prediction in Weighted Signed Networks," in *2016 IEEE 16th International Conference on Data Mining* (ICDM), Barcelona, Spain, December 12–15, 2016 (Piscataway, NJ: Institute of Electrical and Electronics Engineers, 2017), 221–30.

3 Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and V.S. Subrahmanian, "REV2: Fraudulent User Prediction in Rating Platforms," in *WSM '18: Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* , Marina del Rey, California, February 5–9, 2018 (New York: ACM, 2018), 333–41.

was captured at `1289241911.72836` epoch time, or Monday, November 8, 2010, at 13:45 GMT.

> The original source data has time in epoch. The data that accompanies this book uses the ISO 8601 standard because we converted the timestamps for ease of understanding in our examples. For example, `1289241911.72836` in epoch time converts to *`2010-11-08T13:45:11.728360Z`* in the ISO 8601 standard.

Before we can build out interesting queries and a data model, let's take a tour of the world of Bitcoin terminology.

## A Brief Primer on Bitcoin Terminology

Bitcoin is a cryptocurrency, which is used as a decentralized digital currency, meaning there is no central bank or institution that controls its value. Instead, Bitcoin is exchanged on a peer-to-peer network.

Each bitcoin is basically a computer file stored in a digital wallet application on a smartphone or computer. People can send whole bitcoins or fractions thereof to your digital wallet, and you can send bitcoins to other people. Every transaction is recorded in a public list called the blockchain.

*Address*
: An address is a Bitcoin public key to which transactions can be sent.

*Wallet*
: A wallet is a collection of private keys that correspond to addresses.

In our data, we are working with what we can observe on the blockchain. We can observe the exchange of bitcoins between two people. We say you send bitcoins to or receive bitcoins from an address. You encrypt, export, back up, and import your wallet. A wallet can have multiple private keys that correspond to addresses.

From here, we are able to define a schema to use in development for our example.

## Creating Our Development Schema

Though the sample data shows integers, real Bitcoin addresses actually are alphanumeric strings with up to 34 characters. Therefore, we will be using the `Text` data type in our graph schema for the addresses.

The data model we will need is quite simple. We have a list of addresses that rated other addresses. An address can rate another address many times; we would like to capture each rating by its unique rating value.

We talk about the data with the phrase "this address rated that address." Applying our data modeling tips gives us one vertex label, `Address`, and one edge label, `rated`. Figure 8-6 illustrates the conceptual model for our example.

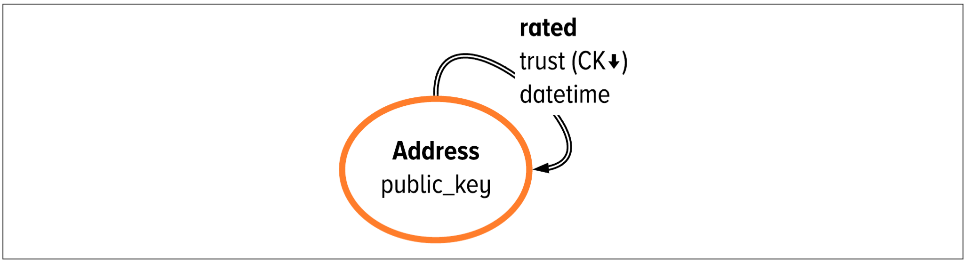



*Figure 8-6. The conceptual model of our graph data*

Using the GSL (graph schema language from Chapter 2), we translate the conceptual model from Figure 8-6 into the schema code in Example 8-2.

*Example 8-2.*

```
schema.vertexLabel("Address").
       ifNotExists().
       partitionBy("public_key", Text).
       create();

schema.edgeLabel("rated").
       ifNotExists().
       from("Address").
       to("Address").
       clusterBy("trust", Int, Desc).
       property("datetime", Text).
       create()
```

Following our setup from Chapter 4, we are again using `Text` as the type for time to make it easier to teach concepts in our upcoming examples. We are using `Text` for time because we will be using the ISO 8601 standard format stored as text: `YYYY-MM-DD'T'hh:mm:ss'Z'`, where `2016-01-01T00:00:00.000000Z` represents the very beginning of January 2016.

Once we have created our graph schema, we are ready to load data.

## Loading Data

We did some basic ETL (extract-transform-load) on `soc-sign-Bitcoinotc.csv` to create two separate files: `Address.csv` and `rated.csv`. This work was required to translate the `datetime` data from epoch into ISO 8601 standard so that the data was ready to be loaded into DataStax Graph.

To get an idea of our data, let's take a look at the top five lines of `rated.csv` in Table 8-1. As before, we set up our `csv` file to have a header. The header line needs to match the names of the properties from your DataStax Graph schema definition in Example 8-2. You can also define a mapping between your `csv` file and database schema when using the loading tool.[4]

*Table 8-1. The first five lines of data from the file `rated.csv`*

| out_public_key | in_public_key | datetime | trust |
|---|---|---|---|
| 1128 | 13 | 2016-01-24T20:12:03.757280 | 2 |
| 13 | 1128 | 2016-01-24T18:53:52.985710 | 1 |
| 2731 | 4897 | 2016-01-24T18:50:34.034020 | 5 |
| 2731 | 3901 | 2016-01-24T18:50:28.049490 | 5 |

From Table 8-1, we can get an idea of the type of data in our example. We will have edges between two public keys, and those edges will have two properties: `datetime` and `trust`. The edge represents a trust rating from one key to another that was created at a certain time and given a rating. For example, let's examine one line of data:

```
|1128|13|2016-01-24T20:12:03.757280|2
```

This line means that the wallet with the key `1128` gave wallet `13` a trust rating of `2` on January 24, 2016, at the time 20:12:04 (rounded).

The accompanying scripts use the same loading process that we have stepped through a few times now. If you would like to see the code, please head to the Chapter 8 data directory within the book's GitHub repository for the data and loading scripts for these examples.

Let's do some basic exploratory queries to ensure that we understand our data and that it loaded correctly.

## Exploring Communities of Trust

The exploration exercise in DataStax Studio observes communities of trust within the data.

We start by confirming that the correct number of vertices and edges have been loaded into our graph. Example 8-3 starts by counting the total number of vertices loaded into DataStax Graph to compare it to the SNAP dataset.

4 See the DataStax Bulk Loader Documentation at *https://docs.datastax.com/en/dsbulk/doc/dsbulk/reference/schemaOptions.html#schemaOptions__schemaMapping*.

*Example 8-3.*

```
dev.V().hasLabel("Address").count()
```

Example 8-3 returns "5881," which matches the total number of unique public keys loaded from the SNAP dataset: 5,881. Next, Example 8-4 counts the total number of edges loaded into DataStax Graph to compare it to the SNAP dataset.

*Example 8-4.*

```
dev.E().hasLabel("rated").count()
```

Example 8-4 returns "35592," confirming the total number of unique ratings from the SNAP dataset: 35,592.

Let's look at a subgraph of trust communities in Figure 8-7, which shows the second neighborhood from a starting address.

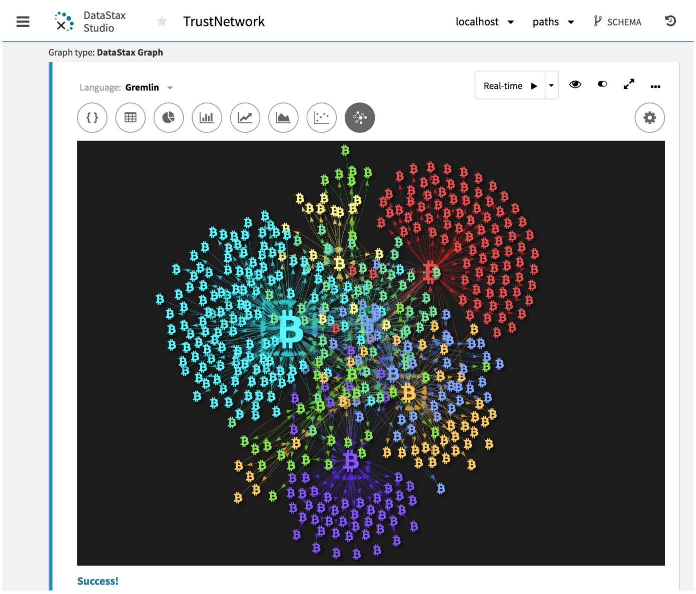



*Figure 8-7. Visualizing a community of trust from a starting public key*

DataStax Studio uses modularity maximization via the Louvain Community Detection Algorithm to assign colors to the subgraph within the Studio client application.

Figure 8-7 displays the second neighborhood from a single starting vertex. We turned on DataStax Studio's graph visualization option to display a graph view of the results and configured the visualization to show community detection within this subgraph.

As we show in Figure 8-7, exploring graph data can be very fun. By creating a simple schema and using bulk loading tools, we hope you were able to follow along from schema creation to data loading and graph visualizations in a matter of minutes.

From here, we want to move away from data exploration and into defining our queries. Our objective is to quantify trust between two wallets by finding the shortest path from one public key to another in this dataset.

# Understanding Traversals with Our Bitcoin Trust Network

Our main objective is to find a good pair of addresses that we'll use in our pathfinding examples in the next section. For the first address in our pair, we cheated a bit. We just randomly selected a starting address: `public_key: 1094`. The interesting work in this section queries the neighborhoods around `1094` to find a good candidate for pathfinding queries. For our purposes, we will be looking for an address that has not previously transacted with `1094` but has many shared connections.

We are constructing a pair of vertices so that we can validate our longer queries later. We admit that this makes our example feel concocted, but we are weaving in practices of test-driven development to illustrate how to test a new Gremlin query for valid and expected results.

Let's start by identifying the addresses that `1094` has previously rated.

## Which Addresses Are in the First Neighborhood?

The addresses in the first neighborhood of `1094` are the same as the addresses that `1094` has previously rated. Example 8-5 reviews how to explore the first neighborhood in Gremlin:

*Example 8-5.*

```
dev.V().has("Address", "public_key", "1094").
    out("rated").
    values("public_key")
```

There are 31 unique addresses in the results of Example 8-5. The first 5 of them are:

```
"1053", "1173", "1237", "1242", "1268",...
```

The 31 addresses in the first neighborhood would not be good candidates for our example because they have a distance of 1 from `1094`, like what you see in Figure 8-8.

Let's move into the second neighborhood.

## Which Addresses Are in the Second Neighborhood?

From the first neighborhood, we need to walk out one more edge to reach the second neighborhood. Example 8-6 shows how to walk to the second neighborhood in Gremlin.

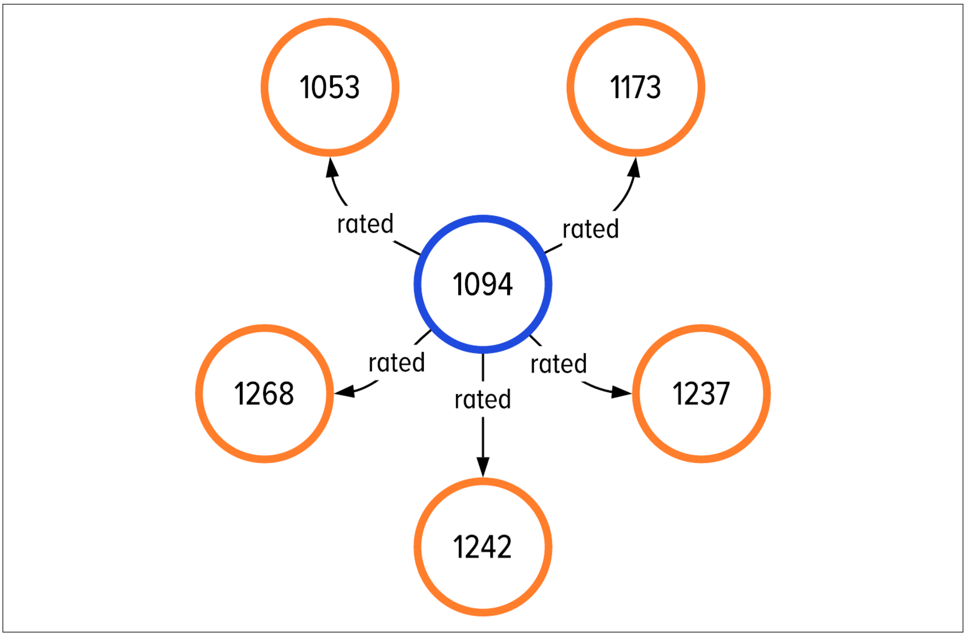



*Figure 8-8. Visualizing some of the addresses in the first neighborhood (a star graph) of `1094`*

*Example 8-6.*

```
dev.V().has("Address", "public_key", "1094").
    out("rated").
    out("rated").
    dedup().      // remove duplicates to get the list of unique neighbors
    values("public_key")
```

There are 613 unique addresses. The first 5 are:

```
"1053", "1173", "1162", "1334", "1241", ...
```

You may be wondering why we needed the `dedup()` step in Example 8-6. We have to use `dedup()` because we want the unique set of addresses in the second neighbor-

hood. Without `dedup()`, the query returns 876 results; those additional 263 results represent multiple ways to walk out two edges from `1094`.

To see this, consider Figure 8-9.

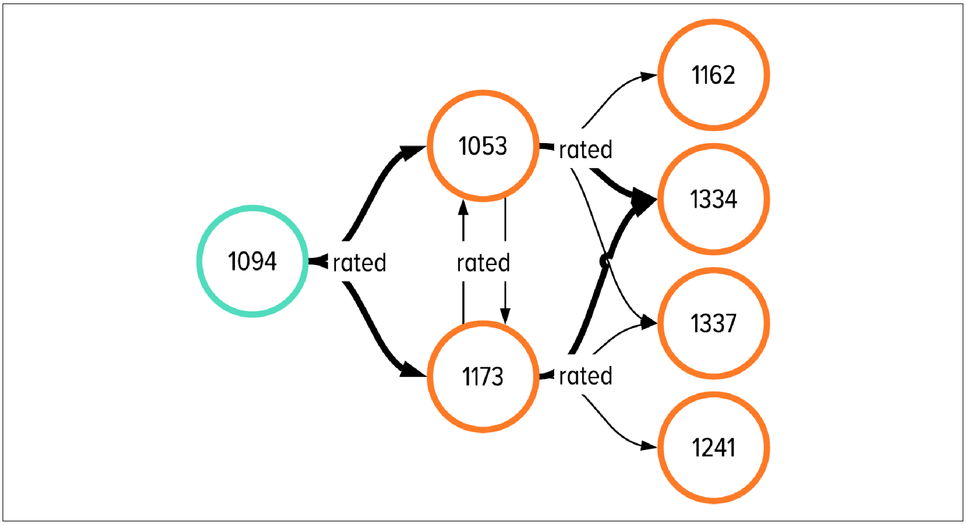



*Figure 8-9. Showing why you need to remove duplicates from the traversal stream*

The address with `public_key 1334` has two different ways to reach `public_key 1094`: via the `1053` or the `1173` vertex. Therefore, `public_key 1334` will be listed at least twice in the second neighborhood of `1094`. Using `dedup()` removes duplicate objects in the traversal stream. For Example 8-6, it takes all observations of `1334` and reduces them to just 1 in the result set.

Using `dedup()` shows how we arrived at a result set size of `613` instead of `876`.

What we really want for our example, however, is an address that is in the second neighborhood but not the first. Let's take a look at how we can find that set of objects in Gremlin.

## Which Addresses Are in the Second Neighborhood, but Not the First?

Before we dive into the query, let's think about what we are asking for here. To see that, let's go back to our sample graph data that shows part of the second neighborhood of `1094`, as seen in Figure 8-10.

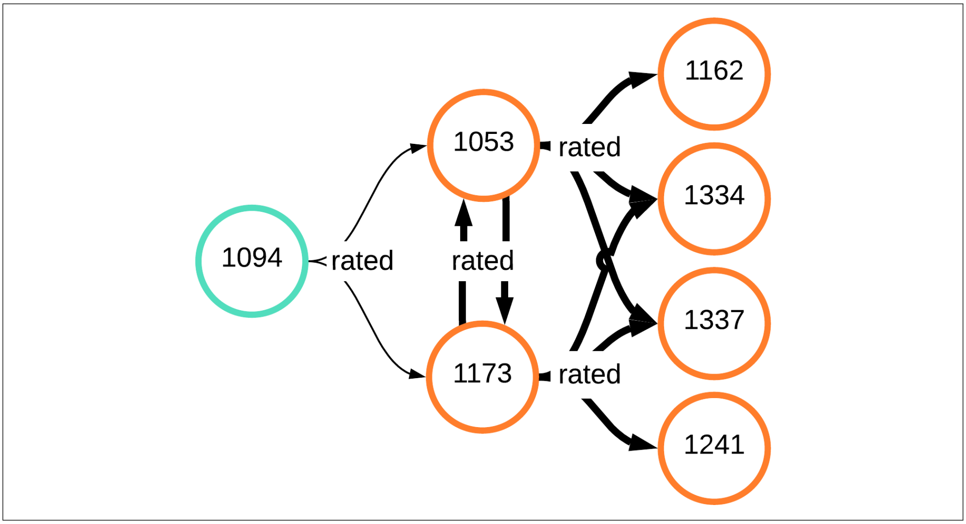



*Figure 8-10. An example of how elements from the first neighborhood of `1094` can also belong to the second neighborhood of `1094`*

Figure 8-10 shows how vertices `1053` and `1173` are members of both the first and the second neighborhood of `1094`. Our original question is looking for a good example that is not directly connected to `1094`. We need to eliminate vertices such as `1053` and `1173` from our result set.

In Gremlin, we can use `aggregate("x")` step to fill an object, named `x` in this example. Then we can eliminate the unwanted vertices from the result set with the `where(without("x"))` pattern. Let's see this in action in Example 8-7.

*Example 8-7.*

```
1 dev.V().has("Address", "public_key", "1094").aggregate("x").
2       out("rated").aggregate("x").
3       out("rated").
4       dedup().
5         where(without("x")).
6       values("public_key")
```

The result set for Example 8-7 has 590 unique elements. The first 5 are:

```
"628", "1905", "1013", "1337", "3062"...
```

Let's walk through Example 8-7. On line 1, we query for `1094` and initialize an object, `x`, with that vertex. On line 2, we traverse to the first neighborhood and add all of those vertices into `x`. Then we walk out to the second neighborhood on line 3.

Let's talk in detail about what is happening between lines 4 and 5 in Example 8-7. The use of `dedup()` on line 4 forces all traversers to complete their work before moving to line 5. In this case, we are waiting for all traversers to reach the second neighborhood away from `1094` before we continue. Then on line 5, we apply a filter with the `where(without("x"))` pattern. Line 5 is essentially asking every traverser in the pipeline, "Are you in the set `x`?" If a vertex is in `x`, it is removed from the pipeline. If not, the traverser is allowed to continue.

> If you have a strong background in relational systems, Example 8-7 is very similar to performing a right outer self join on the address table.

With Example 8-7 in mind, let's take a side tour into the differences between lazy and eager evaluation in the Gremlin query language. We need to dig into the evaluation strategies of the Gremlin query language because they change the behavior of your traversal, which in turn can produce unexpected query results.

> We are about to go really deep into functional programming. If you don't fully understand the next section, it is OK. You just need to get the big-picture point: that barrier steps affect BFS-like and DFS-like behavior in pathfinding with Gremlin.

## Evaluation Strategies with the Gremlin Query Language

Gremlin is primarily a lazy stream-processing language. This means that Gremlin tries to process any traversers all the way through the traversal pipeline before getting more data from the start of the traversal. This is different from an eager evaluation strategy, which does the work right away before moving on to the next step.

*Lazy evaluation*
: Lazy evaluation delays the evaluation of an expression until its value is needed.

*Eager evaluation*
: Eager evaluation evaluates an expression as soon as it is bound to a variable.

There are numerous situations in which the Gremlin language cannot use lazy evaluation.

We are talking about this concept now because we have been using eager evaluation in our recent traversals with the `dedup()` and `aggregate()` steps.

> In your daily life, you may choose to perform a task with either evaluation strategy. You probably use eager evaluation when you are cooking because you prepare each ingredient of your meal and then assemble individual plates. Contrast this with creating plate-sized portions that you cook individually from beginning to end.

In Gremlin, the key to knowing when a traversal changes between lazy and eager evaluation is to recognize the barrier steps. When a barrier step exists, a Gremlin traversal changes from lazy to eager evaluation.

#### Barrier steps in Gremlin

The definition of a barrier step in the Gremlin query language is:

*Barrier step*
: A barrier step is a function that turns the lazy traversal pipeline into a bulk-synchronous pipeline.

We want to make this distinction about barrier steps because the Gremlin query language mixes the use of lazy evaluation strategies and eager evaluation strategies when there are barrier steps.

Barrier steps change the behavior of a query to operate like breadth-first search or depth-first search.

Examples of barrier steps used in this book are `dedup`, `aggregate`, `count`, `order`, `group`, `groupCount`, `cap`, `iterate`, and `fold`.

One way to think about these concepts together is that barrier steps force a pipeline to execute like breadth-first search. That is, barrier steps force every traverser to wait until all other traversers in the pipeline have completed the same set of work. After all traversers complete the work up to a barrier step, they can continue.

The queries we demonstrate in this book aim to teach the common patterns found in real-time applications. As such, we are mixing BFS and DFS behavior as we write our queries.

We will apply the connection between barrier steps and BFS in a later example to guarantee shortest paths in our queries.

## Pick a Random Address to Use for Our Example

Previously, we successfully found the vertices that are in the second neighborhood, but not the first. Now let's use the `sample()` step, as demonstrated in Example 8-8, to randomly select one of them to use for the rest of our queries.

*Example 8-8.*

```
dev.V().has("Address", "public_key", "1094").aggregate("first_neighborhood").
    out("rated").aggregate("first_neighborhood").
    out("rated").
    dedup().
      where(neq("first_neighborhood")).
    values("public_key").
    sample(1)
```

The result is:

```
"1337"
```

Believe it or not, `1337` was the first `public_key` we randomly sampled for the rest of our pathfinding examples. We are going to take that as a good omen and go with it.

Now we have two addresses: `1094` and `1337`. Let's use them to show how to find paths between them with Gremlin.

# Shortest Path Queries

As we mentioned earlier, we selected this dataset and example to illustrate the power of using paths to solve complex problems. Distance between concepts or people provides context and meaning for assessing how related they are and whether you can trust them.

For the rest of these exercises, we want you to imagine you have joined a Bitcoin marketplace, specifically the Bitcoin OTC. When you joined, you received the public key `1094`. Think about your first transaction on that marketplace with a member who has the public key of `1337`.

How much trust do you have in the other address?

Quantifying your trust in another entity with path analysis is the complex question we are going to determine in the next series of exercises.

The upcoming example has five main sections.

The first section begins with finding paths of fixed lengths between our example addresses. We will build upon the queries from the first section to find paths of any length in the second section. The second section illustrates a common progression through applying pathfinding techniques, but it purposefully leads to an error.

The third section explains how we can resolve our error by revisiting lazy and eager evaluation with Gremlin. The fourth section explores understanding path weight for the shortest paths in our example data.

We will conclude with a discussion of how to interpret path length and context for quantifying trust to our question. This sets up how we will transform this dataset to find weighted shortest paths in Chapter 9.

## Finding Paths of a Fixed Length

We are starting with finding paths of a fixed length by exploring neighborhoods so that we can validate the results that show up in our shortest path queries at the end of this section.

To get into the mindset of these walks, consider what you would want to know before you accepted someone's invitation to exchange bitcoins.

If you were about to transact with a new address on a Bitcoin OTC marketplace, you would likely want to know whether you can trust the other person. The place to start in quantifying your trust in `1337` is to find out whether you have shared connections. Finding shared connections in this dataset is the same as looking for addresses you rated that also rated `1337`, or vice versa. This type of shared connection doesn't care about the direction of the rating; we just want to see what shared addresses we have according to who rated whom.

One way to do this is to count the number of ways you can reach `1337` in your second neighborhood. Let's do this query in Example 8-9.

*Example 8-9.*

```
dev.V().has("Address", "public_key", "1094").as("start").
        both("rated").
        both("rated").
        has("Address", "public_key", "1337").
        count()
```

The result of Example 8-9 is 4. This means that within your second neighborhood, there are four ways to walk from your address, `1094`, to `1337`. Let's look at the path information to understand those walks.

Recalling our discussion from Chapter 6, the `path()` step will give you access to each traverser's full history. Then you want to look at the results according to two features: (1) the vertices visited along the way and (2) the path's length. Let's do this in Example 8-10 and then walk through the process and results.

*Example 8-10.*

```
dev.V().has("Address", "public_key", "1094").
        both("rated").
        both("rated").
        has("Address", "public_key", "1337").
```

```
5        path().                                  // traverser's full path history
6          by("public_key").as("traverser_path"). // get each vertex's public key
7        count(local).as("total_vertices").       // count the elements in the path
8        select("traverser_path", "total_vertices")  // select the path information
```

Let's step through the query in Example 8-10 before we look at the results in Example 8-11.

Lines 1 through 3 in Example 8-10 walk to the second neighborhood from `1094`. Line 4 considers only those walks that ended at `1337`. Then we want to get the path information from each of the four traversers via the `path()` step on line 5. Line 6 mutates the path objects to show only their `public_key` and stores a reference to it. Then on line 7, we count the total number of objects within each path with `count(local)`. Here, the local scope asks to count the total number of objects within the path instead of using the default global scope of `count()`, which would count the total number of paths. On line 8, we select each path object alongside the total number of vertices within each path.

The results are shown in Example 8-11.

*Example 8-11.*

```
{
  "traverser_path": { "labels": [[],[],[]],
                      "objects": ["1094", "1268", "1337"]},
  "total_vertices": "3"
},{
  "traverser_path": { "labels": [[],[],[]],
                      "objects": ["1094", "1268", "1337"]},
  "total_vertices": "3"
},{
  "traverser_path": { "labels": [[],[],[]],
                      "objects": ["1094", "1268", "1337"]},
  "total_vertices": "3"
},{
  "traverser_path": { "labels": [[],[],[]],
                      "objects": ["1094", "1268", "1337"]},
  "total_vertices": "3"
},
```

The path object has very useful information. We see that we only really share address `1268` in common. There are four paths because there were four different combinations of how `1094` or `1337` rated `1268`. If you would like, you could confirm this for yourself by inspecting the edges along the paths. But we are going to move on to the next query.

It has been helpful to find any path in our second neighborhood to `1337`.

However, let's start looking deeper into the data and consider only one direction: `out()`. Specifically, we want to know: how many outgoing paths to `1337` can we find in our third neighborhood? Further, let's simplify this query by using the `repeat().times(x)` pattern to discover the paths in our third neighborhood in Example 8-12.

*Example 8-12.*

```
dev.V().has("Address", "public_key", "1094").  // start at 1094
       repeat(out("rated")).                    // walk out "rated" edges
       times(3).                                // three times
       has("Address", "public_key", "1337").   // until you reach 1337
       path().                                  // get the path of each traverser
         by("public_key").as("traverser_path").// for each path, get vertex's key
       count(local).as("total_vertices").      // count the number of objects
       select("traverser_path", "total_vertices") // select the path, length
```

The first three results are shown in Example 8-13.

*Example 8-13.*

```
{
  "traverser_path": { "labels": [[],[],[],[]],
                      "objects": ["1094","1268","35","1337"]},
  "total_vertices": "4"
},{
  "traverser_path": { "labels": [[],[],[],[]],
                      "objects": ["1094","280","35","1337"]},
  "total_vertices": "4"
},{
  "traverser_path": { "labels": [[],[],[],[]],
                      "objects": ["1094","1053","1268","1337"]},
  "total_vertices": "4"
},...
```

We can see some interesting paths from `1094` to `1337` in Example 8-13. The third result shows that `1094` rated `1053`, who rated `1268`, who rated `1337`. There are 11 total outgoing paths in our third neighborhood from `1094` to `1337`.

We can generalize the `repeat().times(x)` pattern to find paths of a known length. However, our overarching goal is to find paths of any length to eventually discover how to use Gremlin to discover shortest paths.

## Finding Paths of Any Length

Pathfinding queries are for discovering the relationships that connect two things in your graph together. We want to discover both the quantity and the depth of the relationships that exist in the data.

That is, instead of querying for paths of a known length, we want to find paths of any length.

More often than not, we see engineers make the leap from paths of defined length to paths of unbounded length with queries like what we have in Example 8-14. As you see in Table 8-2, this is likely going to lead to an execution error.

*Example 8-14.*

```
1 dev.V().has("Address", "public_key", "1094").        // start at 1094
2         repeat(out("rated")).                        // walk out rated edges
3         until(has("Address", "public_key", "1337")). // WARNING: this is all-paths!
4         path().
5           by("public_key").as("traverser_path").
6         count(local).as("total_vertices").
7         select("traverser_path", "total_vertices")
```

If you ran Example 8-14 in DataStax Studio, you most likely saw the error shown in Table 8-2:

*Table 8-2. An example of a system error due to a traversal taking longer than 30 seconds*

| System error |
| --- |
| Request evaluation exceeded the configured threshold of realtime_evaluation_timeout at 30000 ms for the request |

Let's walk through what is happening in Example 8-14 so that we can understand the error from Table 8-2. Line 1 in Example 8-14 accesses the starting address, `1094`. Lines 2 and 3 apply the `repeat().until()` pattern. The `repeat()` step tells a traverser what it is supposed to do until the breaking condition from the `until()` step. We have just asked our traversers to keep searching for any path that starts at `1094` and ends at `1337`. This is going to explore the entire connected graph for all paths that end at `1337`. This is why we get the timeout error in Table 8-2.

For our problem we do not want all paths. We want to find the shortest path. Let's connect some concepts together to try a different approach.

### Connecting concepts: BFS and traversal strategies

Recall our discussion from "Evaluation Strategies with the Gremlin Query Language" on page 244. We went through evaluation strategies, barrier steps, and thinking through how breadth-first or depth-first searching applies to traversals.

We told you we were going to apply those facts to find shortest paths. Let's do that now.

We need to figure out whether our pathfinding traversal is using BFS or DFS. If it is using BFS, then we can guarantee that the first traverser that satisfies the stopping condition is the shortest path.

Thinking in Gremlin, traversals that are eagerly evaluated provide the behavior we need to guarantee BFS behavior. The key to figuring out if your traversal uses eager evaluation is to find out whether its execution strategy uses barrier steps.

Looking at our traversal from Example 8-14, the query did not use any of the barrier steps we talked about before. What are we missing?

The definitive way to answer this for yourself is to inspect the `explain()` step to see what traversal strategies are applied, as we have done in the query in Example 8-15.

*Example 8-15.*

```
g.V().has("Address", "public_key", "1094").
      repeat(out("rated").
      until(has("Address", "public_key", "1337")).
      explain()

==>Traversal Explanation

Original Traversal   [GraphStep(vertex,[]),
                      RepeatStep([VertexStep(OUT,vertex),
                      RepeatEndStep],until(),emit(false))]
...
Final Traversal      [TinkerGraphStep(vertex,[]),
                      VertexStep(OUT,vertex),
                      NoOpBarrierStep(),     // Note: Barrier Execution Strategy
                      VertexStep(OUT,vertex),
                      NoOpBarrierStep(),     // Note: Barrier Execution Strategy
                      VertexStep(OUT,vertex),
                      NoOpBarrierStep()]     // Note: Barrier Execution Strategy
```

The `explain()` step prints out the traversal explanation for your traversal. A traversal explanation details how the traversal (prior to `explain()`) will be compiled given the registered traversal strategies.

Looking at Example 8-15, we see something very interesting: `NoOpBarrierStep`. The presence of `NoOpBarrierStep` in the traversal explanation informs us that the traversal engine injects barrier steps with the `repeat()` step.

We use the information from Example 8-15 to know that the `repeat().until()` pattern uses barriers. This means it executes eagerly using breadth-first search.

With one small change to Example 8-14, we can apply this knowledge in Example 8-16, which finds the single shortest path from `1094` to `1337`.

*Example 8-16.*

```
1 dev.V().has("Address", "public_key", "1094").      // start at 1094
2       repeat(out("rated")).                         // walk out rated edges
3       until(has("Address", "public_key", "1337")). // until 1337
4       limit(1).           // BFS: the first traverser the shortest path
5       path().             // get the traverser's path information
6         by("public_key").as("traverser_path").    // get each vertex's public_key
7       count(local).as("total_vertices").         // count each path's length
8       select("traverser_path", "total_vertices")  // select the path information
```

The important line to understand in Example 8-16 is line 4. The `limit(1)` step passes only one traverser into the remaining pipeline. Because the `repeat().until()` steps are eagerly evaluated, we can guarantee that the first traverser to satisfy the stopping condition is also the shortest path!

The path object for this traverser is:

```
{
  "traverser_path": { "labels": [[],[],[]],
                      "objects": ["1094", "1268", "1337"]},
  "total_vertices": "3"
}
```

This confirms what we already knew from the examples we did a while back: the shortest path from `1094` to `1337` is through `1268`. We spent so much time setting up this example and walking through paths of fixed length so that when we got here, we could confirm that the path we found was indeed the shortest.

Zooming back out a bit, let's think about how we would want to apply this information to answer this section's main question. We have discovered you have one address in common: `1268`. We also know that there are 11 ways we can find friends of friends that you have in common with `1337`, which is the same as saying there are 11 paths of length 3 between you and `1337`.

If you were really trying to make a decision about transacting with `1337`, would you have enough information? Would you trust this address?

Maybe you want to understand the types of ratings that were given on these paths. Let's look at three final queries to start to quantify trust from our edges onto our paths.

## Augmenting Our Paths with the Trust Scores

The next piece of information you likely want to consider is the trust ratings in the data along all of these paths. To look at those, we will want to reformat the data structure for our queries. Example 8-17 expands our shortest path query from Example 8-16 in two ways. First, it applies our knowledge of BFS and Gremlin query processing to find the top 15 shortest paths from `1094` to `1337`. Then, it reformats the results using the `project()` step. Let's take a look at the query and its results.

*Example 8-17.*

```
1 dev.V().has("Address", "public_key", "1094").
2       repeat(out("rated")).
3       until(has("Address", "public_key", "1337")).
4       limit(15).  // BFS: return the first 15 shortest paths by length
5       project("path_information", "total_vertices").
6         by(path().by("public_key")).
7         by(path().count(local))
```

*Example 8-18.*

```
{
  "path_information": { "labels": [[],[],[]],
                        "objects": ["1094","1268","1337"]},
  "total_vertices": "3"
},{
  "path_information": { "labels": [[],[],[],[]],
                        "objects": ["1094","280","35","1337"]},
  "total_vertices": "4"
},{
  "path_information": { "labels": [[],[],[],[]],
                        "objects": ["1094","1268","35","1337"]},
  "total_vertices": "4"
},...
```

The main work of Example 8-17 is on lines 4 through 7. On line 4, we are taking only the first 15 traversers that reach the stopping condition from line 3. These first 15 traversers are guaranteed to be the 15 shortest paths because of how Gremlin uses barrier steps to process graph data like breadth-first search.

Then, starting on line 5 of Example 8-17, you see how we are going to format our results for the remaining queries in this chapter. We want to create a map with keys and values. The keys in the map will be `path_information` and `total_vertices`. The

`by()` modulator on line 6 fills in the `path_information` key with a formatted version of the `path()` object from `1094` to `1337`. The `by()` modulator on line 7 fills in the `total_vertices` key with the `public_key` of each visited vertex on the path from `1094` to `1337`.

#### Using sack() to aggregate trust ratings

Let's add one more key to the map from Example 8-17. Let's add up the `trust` values on the `rated` edges along the way and add this key/value pair to our results set. Adding up the `trust` values for each edge will represent the total trust of the path from `1094` to `1337`.

As you walk through graph data, we will need a way to aggregate information that we process along the way. The `sack()` step in Gremlin gives us this ability.

You can think of the `sack()` step as giving your Gremlin traverser a backpack at the start of its journey through the data. Along the way, you tell your traverser what to add or remove from the backpack (sack). This is very useful for collecting values on vertices or edges along the way and using them to make decisions or collect metrics.

For our paths, we want to add up the trust ratings from the edges. We will be giving our traverser an empty sack to start and then augmenting its contents with the trust ratings as it walks over edges. Figure 8-11 shows how this works conceptually.

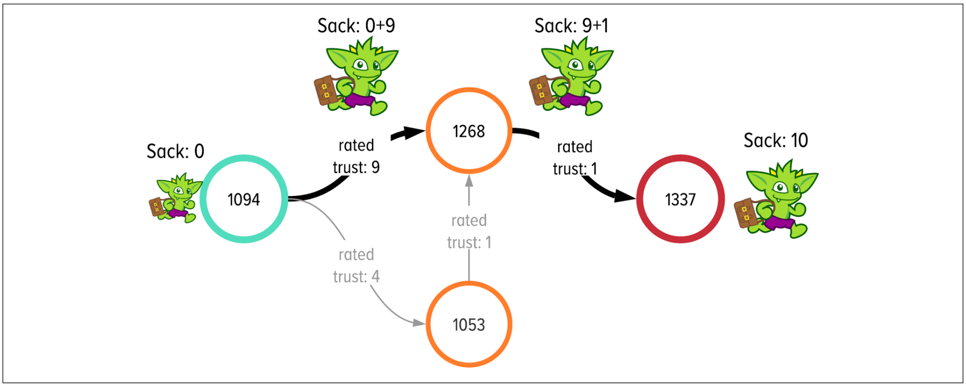



*Figure 8-11. Showing how a traverser moves from the start, `1094`, through a path to the end, `1337`, and stores the trust ratings on edges in its sack along the way*

In Figure 8-11, the traverser walks the shortest path from `1094` to `1337`: a path of length 2 via the `1268` vertex. We show how we can use the `sack()` object to collect and aggregate the trust ratings from the edges during the traversal. The ending `sack()` value for this path is 10.

There is a second path a traversal can explore in Figure 8-11. This longer path that traverses through the `1053` vertex is shown in Figure 8-12.

The definitions for sack constructs in Gremlin are:

- A traverser can contain a local data structure called a `sack`.
- The `sack()` step is used to read and write sacks.
- Each sack of each traverser is initialized when using `withSack()`.

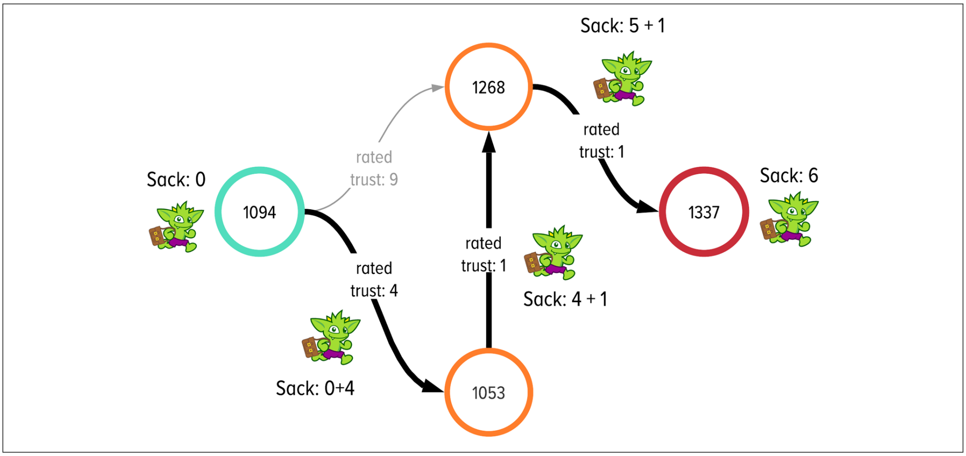



*Figure 8-12. Showing how a traverser moves from the start, `1094`, through a different path to the end, `1337`, and stores the trust ratings on edges in its sack along the way*

Revisiting our query, we can calculate the total trust for our 15 shortest paths. We now know that we will use the `sack()` step to add up the trust ratings for each path. We also want to add this data as a new key in our result payload. The key `total_trust` will be our new key in the `project()` step. The value for `total_trust` will be the sum of the edge weights along the path using the `sack()` step.

Let's see how we do this in Gremlin in Example 8-19.

*Example 8-19.*

```
dev.withSack(0.0).       // initialize each traverser with a value of 0.0
  V().
  has("Address", "public_key", "1094").as("start").
  repeat(outE("rated").  // walk out and stop on the "rated" edge
         sack(sum).      // add to the traverser's sack
           by("trust").  // the value from the property "trust"
         inV()).         // leave the edge and walk to the incoming vertex
  until(has("Address", "public_key", "1337")).  // repeat until 1337
  limit(15).             // limit to the 15 shortest paths by length
```

```
10 project("path_information", "vertices_plus_edges", "total_trust").  // a map
11    by(path().by("public_key").by("trust")).   // first value: path information
12    by(path().count(local)).                    // second value: path's length
13    by(sack())                                  // third value: path's trust score
```

Let's step through Example 8-19. Line 1 shows how to initialize your traversal to use a local data structure for each traverser in your query: `withSack(0.0)`. The next section to really dig into is lines 4 through 8. On line 4 and line 8, we see the expected `repeat()/until()` pattern for walking through our graph data for pathfinding with breadth-first search. Notice, however, that line 4 uses the `outE()` step. Using `outE()` ensures that each traverser stops on the edge between two vertices. It is necessary to stop on edges so we can collect the trust rating. Then on line 5, we tell the traverser to add something into its sack via `sack(sum)`. You use `by()` modulators to tell the sack what you are adding into it. You find the `by()` modulator on line 8: `by("trust")`. The pattern of `sack(sum).by("trust")` tells the traverser to collect the `trust` property from its current object, which is an edge, and to add it to the value currently in its sack.

Then we tell the traverser to move to the incoming vertex with `inV()` on line 7. The stopping condition on line 8 asks a traverser to repeat this behavior until it reaches `1337`. The first 15 traversers that meet this condition continue down into the `project()` step on line 10. Line 10 formats our results into a hashmap. The first key and value pair in the hashmap formats the `path` object to alternate from the vertex's public key and the edge's `trust` value, respectively. The second key and value pair in the hashmap counts the total number of objects in the path object. Because we visited edges along the way at line 4, we will have both vertices and edges in our path object. Therefore, we expect the total calculated on line 12 to be the sum of vertices and edges along the way.

Last, line 13 in Example 8-19 tells each traverser to report its sack's value as we read its contents with `sack()`. The full table of results from Example 8-19 is shown in Example 8-20.

*Example 8-20.*

```
{
  "path_information": {
       "labels": [[],[],[],[],[]],
       "objects": ["1094","9","1268","1","1337"]},
  "vertices_plus_edges": "5",
  "total_trust": "10.0"
},{
  "path_information": { "labels": [[],[],[],[],[],[],[]],
                        "objects": ["1094","4","1053","1","1268","1","1337"]},
  "vertices_plus_edges": "7",
  "total_trust": "6.0"
```

```
},{
  "path_information": { "labels": [[],[],[],[],[],[],[]],
                        "objects": ["1094","9","1268","1","35","9","1337"]},
  "vertices_plus_edges": "7",
  "total_trust": "19.0"
},...
```

It would be most interesting to look at the paths with the most trust. Let's add some sorting to the results from Example 8-19 to display our 15 shortest paths, sorted by their total trust. After we have our 15 shortest paths and before we format them, we just need to apply the sorting logic. You see this on lines 10 and 11 in Example 8-21.

*Example 8-21.*

```
1 dev.withSack(0.0).
2   V().
3   has("Address", "public_key", "1094").
4   repeat(outE("rated").
5          sack(sum).
6            by("trust").
7          inV()).
8   until(has("Address", "public_key", "1337")).
9   limit(15).
10  order().               // order all 15 paths
11    by(sack(), decr).    // according to each traverser's sack value, decreasing
12  project("path_information", "vertices_plus_edges", "total_trust").
13    by(path().by("public_key").by("trust")).
14    by(path().count(local)).
15    by(sack())
```

The sorting logic on lines 10 and 11 in Example 8-21 globally arranges, in decreasing order, the 15 traversers in the pipeline according to the value within each traverser's sack. The first result is shown in Example 8-22.

*Example 8-22.*

```
{
  "path_information": { "labels": [[],[],[],[],[],[],[],[],[]],
                        "objects": ["1094","9","1268","10","1094","9","1268","1",
                           "1337"]},
  "vertices_plus_edges": "9",
  "total_trust": "29.0"
},...
```

Did you notice something unexpected in Example 8-22? The highest weighted path has two cycles between `1094` and `1268`. This type of path wouldn't make sense in our application because we are considering the ratings between two keys more than once.

We introduced and used `simplePath()` in Chapter 6 to remove cycles; let's add that step here. Adding `simplePath()` adjusts our final query to find the 15 shortest paths without cycles and then sorts the 15 paths by their aggregated trust score, in descending order. Example 8-23 shows our final query, and Example 8-24 displays the results.

*Example 8-23.*

```
dev.withSack(0.0).
  V().
  has("Address", "public_key", "1094").
  repeat(outE("rated").
         sack(sum).
           by("trust").
         inV()
         simplePath()).    // remove a traverser if there is a cycle in its path
  until(has("Address", "public_key", "1337")).
  limit(15).
  order().
    by(sack(), decr).
  project("path_information", "vertices_plus_edges", "total_trust").
    by(path().by("public_key").by("trust")).
    by(path().count(local)).
    by(sack())
```

Example 8-24 shows the top 15 shortest paths without cycles, sorted by their aggregated trust score.

*Example 8-24.*

```
{
  "path_information": {"labels": [[],[],[],[],[],[],[],[],[]],
                       "objects": ["1094","10","64","10","104","3","35","9","1337"]},
  "vertices_plus_edges": "9",
  "total_trust": "32.0"
},{
  "path_information": {"labels": [[],[],[],[],[],[],[],[],[]],
                       "objects": ["1094","9","1268","2","1201","5","35","9","1337"]},
  "vertices_plus_edges": "9",
  "total_trust": "25.0"
},{
  "path_information": {"labels": [[],[],[],[],[],[],[],[],[]],
                       "objects": ["1094","3","280","8","35","9","1337"]},
  "vertices_plus_edges": "7",
  "total_trust": "20.0"
}...
```

The query in Example 8-23 brings together all of the path concepts to explore in development. We use our knowledge of breadth-first search in Gremlin to find the 15 shortest paths and augment our results with each path's weight. The top three highest

weighted paths are shown in Example 8-24. Our results show that the longer the path, the higher its weight.

Are the results in Example 8-24 what you would want to use for determining whether you trust address `1337`?

You are likely shouting "No!" The results in Example 8-24 show that the paths with the most trust value are also the longest paths. Longer walks through our data will aggregate more trust ratings along the way and therefore are "more trusted."

The structure of our data and pathfinding queries in development are not returning results that make sense for an application.

> You may see different results from Example 8-24 in your Studio Notebook. This is because the top 15 paths include three paths of length 4 (nine total objects: five vertices, four edges). There are more than three paths of length 4; and the results will include the first three that are discovered.

Our exploration in development has left us with two optimizations we need to address for a production-quality query. First, we need a different way to understand and use weights in making our decision about trust. The way that `trust` is represented in the dataset now is not providing results that are meaningful to a user of this data.

The second optimization we need is to find paths that are both short and with high trust. In development, we discovered that our tools can find either shortest paths by length or all paths. And it is too expensive to find all paths. We need a different approach for shortest weighted paths for our production-quality queries.

We need to normalize the edge weights on this data so that we can properly find shortest weighted paths. That is the theme and objective of the next chapter.

## Do You Trust This Person?

We opened this chapter with an idea: the idea that humans naturally use distance between concepts as a positive correlation to how much we trust the association between those concepts.

To quantify our idea, we defined the shortest path problem, walked through the fundamentals of searching through graph data, and applied those concepts with the Gremlin query language. Then our development examples showed how using paths to quantify trust in a network informs a decision about transacting on the Bitcoin OTC network.

However, we realized that we cannot simply add up trust scores as a measure of trust in this network to quantify our most valuable paths. To discover the most trusted paths in our data, we need to introduce two concepts for production use of pathfinding: normalization and query optimizations.

Continue with us to the next chapter to learn how teams commonly evolve their thinking to address a more complex problem in production: shortest weighted paths in graph data.

# CHAPTER 9
# Finding Paths in Production

More often than not, the first concept we think about with paths is how many stops it takes to get from the start to the finish. This was the topic for Chapter 8.

The next concept when working with paths through graphs is to evolve the idea of *distance*. We do this by adding some type of weight or cost to steps along a path. We refer to this type of problem as a *minimum cost path* or a *shortest weighted path*.

Shortest weighted paths are very popular optimization problems in computer science and mathematics. These types of problems tend to be multifaceted, complex optimization problems because they are trying to combine more than one source of information into a cost metric for minimization.

We saw an example of a weighted path problem at the end of Chapter 8. We tried to find the most trusted path through our data by aggregating path weights. Because high trust in our sample data is represented by higher values, this type of pathfinding problem led to the discovery that higher trust paths are also longer paths through our data. This is not what we wanted.

Instead, we need to understand how to use edge weights to find shortest paths. Through the lenses of mathematics and computer science, we want to create a bounded minimum optimization problem.

In this sense, high trust is inversely correlated with path length. We want to find paths that are simultaneously short and have high trust. This is the difficult duality we are going to address and optimize in this chapter.

## Chapter Preview: Understanding Weights, Distance, and Pruning

There are three main sections in this chapter.

In the first section, we are going to formally define the shortest weighted path problem and walk through the algorithm. Our pathfinding algorithm uses breadth-first search to optimize pathfinding to find shortest weighted paths.

The second section introduces the edge weight normalization process. We will walk through the general process of shifting and flipping our weights' scale from "higher is better" to "lower is better." We will show the new weights we calculated for our sample dataset, create a new edge, and reload the normalized trust scores for our example.

The last section uses the A* algorithm on our normalized data. We will break down writing A* in the Gremlin query language and run it on our example data to find the shortest weighted paths between your public key `1094` and your open invite with `1337`.

Though your journey through this book has been long, we hope you have high trust in our upcoming examples. See? You are already correlating longer paths with higher trust.

# Weighted Paths and Search Algorithms

We have already tried to use edge weights in our pathfinding problems. We did this at the end of Chapter 8 when we introduced the `sack()` step to aggregate trust ratings across paths in the Bitcoin OTC trust network.

However, our process was inefficient because the tools we had did not solve the problem we thought we were trying to solve. This section addresses two reasons our first attempt didn't work by teaching two new tools.

First, we will define the problem for shortest weighted paths and look at a few correct examples. Then, we will introduce a new algorithm for finding solutions to shortest weighted path problems, the A* search algorithm. You will see these tools later when we build the A* search algorithm in Gremlin to find shortest weighted paths in the normalized Bitcoin OTC network.

Let's get started with a new problem definition.

## Shortest Weighted Path Problem Definition

Recall that in Chapter 8 we defined shortest paths. As a refresher, the shortest path in a graph is the fewest number of edges it takes to walk from one vertex to another in the graph.

A weighted path uses properties from your graph data to aggregate and score a path's weighted distance from start to end. The shortest weighted path is the path with the lowest score:

*Shortest weighted path*
: The shortest weighted path discovers the path between two vertices in a graph such that the total sum of the edges' weights is the minimum.

Let's use a concrete example; Figure 9-1 adds some weights to our example graph from Figure 8-4.

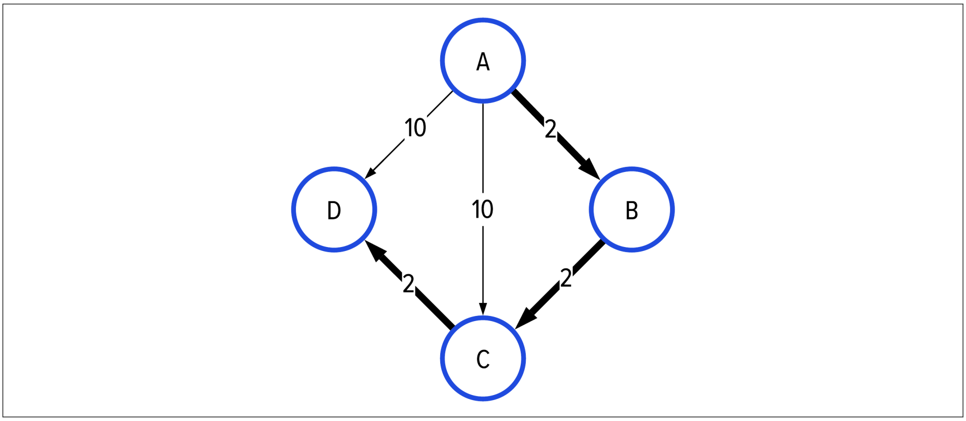



*Figure 9-1. A weighted graph with bolded edges to show the shortest weighted path from `A` to `D`*

Figure 9-1 uses bolded edges to illustrate the shortest weighted path from vertex A to vertex D.

The total weight of the shortest weighted path from `A` to `D` is 6. Contrast this weight with the shortest path in the graph. The shortest path in the graph is `A → D` and has a weight of 10. This path is not the shortest weighted path because A → B → C → D has a lower weight of 6.

With a new problem comes new approaches. In the small example in Figure 9-1, we can quickly see the shortest path.

For larger graphs, we need to fold in multiple optimizations. The only optimization we have so far from BFS and DFS tracks the visited set of vertices so that we do not repeat exploring the same space.

But we can be smarter when we are working with weighted graphs. Let's delve into shortest weighted paths in graph data.

## Shortest Weighted Path Search Optimizations

A quick Google search on "graph path algorithms" returns an extensive list, including A* (pronounced "A star"), Floyd-Warshall, and Dijkstra's, to name a few. We are zooming in on the optimizations that these algorithms apply to teach you the fundamentals that apply to any approach. The costs and benefits of different searching algorithms come from understanding how they reduce the search space with different creative optimizations.

Different algorithms solve the shortest weighted path problem for graphs by applying a few optimizations along the way. A graph search algorithm maintains a tree of paths from the starting vertex and applies heuristics to decide whether a new edge should be added into the working tree. At a high level, some of those optimizations include:

*Lowest cost optimization*
: The lowest cost optimization excludes an edge if the edge's destination is reachable via a lower cost path.

*Supernode avoidance*
: Supernode avoidance excludes a vertex if its degree would increase the search space complexity over a threshold.

*Global heuristic*
: A global heuristic excludes an edge if the edge's weight causes the path's total weight to exceed a threshold.

> There are a myriad of heuristics you can apply to optimize your graph algorithm. Choosing good heuristics requires understanding your data, its distributions, and the graph structures you want to avoid during pathfinding.

The second optimization defined here notes that you could optimize your search space by eliminating supernodes. Let's take a brief side tour to define supernodes and explain why you would want to use a heuristic to remove them from your search space.

#### Supernodes in graphs

The idea of a supernode is that it is a vertex with an extremely high number of edges. That is where the idea of *super* comes from; a supernode is a highly connected vertex in your graph data.

*Supernode*
: A supernode is a vertex with a disproportionately high degree.

For a direct example, think about Twitter's social network. You want to mentally draw out a graph of Twitter accounts in which the edges are who follows whom. A supernode is a vertex with a very high number of followers compared to the rest of the network. Most celebrities on Twitter are good examples of supernodes.

> Fun fact: in the early days of building Apache Cassandra, the team developed counters to track the number of followers for a Twitter account. This was known as the Ashton Kutcher problem, as he was the first to reach 1,000,000 followers on Twitter. The volume of followers makes Ashton Kutcher's account a supernode in the Twitter network.

As it relates to pathfinding, if you traverse into a supernode, you potentially add millions of edges to your priority queue. This will blow up the computational cost of your traversal due to the many new edges to consider for pathfinding.

To this end, let's walk through some theoretical limitations with supernodes.

#### Theoretical limits of supernodes

In Apache Cassandra, a partition can contain, at most, two billion cells. An edge table in DataStax Graph requires the primary keys for each endpoint vertex, therefore requiring a minimum of two cells per edge. But to get unique edges, you need some type of universally unique identifier (UUID) on the edge. Thus, the minimum number of cells in an edge's partition is three: two billion divided by three max cells when you reach the uppermost limit of storing a supernode on disk.

That means that in DataStax Graph, a single vertex with 666,666,666 edges is one edge away from hitting the limit on disk for the number of cells in an Apache Cassandra table. That's ominous.

Regardless, you will hit a snag with processing supernodes in a traversal well before you create one on disk. To see this, think back to the processing limitations we discovered in Chapter 6. We ran into processing limitations due to our graph's branching factor for relatively low degree vertices. It is safe to say that you are likely to be troubleshooting the processing performance of supernodes well before you reach limitations on disk.

Our approach with supernodes in our upcoming implementation will be to eliminate them entirely. Let's outline how we will apply this technique, and a few more optimizations, in the next section.

### Pseudocode for the search algorithm we will implement

Let's first understand the pseudocode approach for the algorithm we are going to build. We will implement a BFS algorithm in Gremlin with optimizations specific to our dataset in a future section.

We are going to apply three optimizations to pathfinding in our weighted Bitcoin OTC network:

- Lowest cost optimization excludes the edge if we have already found a shorter path to the next vertex.
- Supernode avoidance excludes an edge if the destination vertex has too many outgoing edges.
- Global heuristic excludes the edge if the edge's weight causes the path's total weight to exceed the maximum value we want to consider.

The pseudocode in Example 9-1 describes the algorithm that we will be implementing in this chapter.

*Example 9-1.*

```
ShortestWeightedPath(G, start, end, h)
   Use sack to initialize the path distance to 0.0
   Find your starting vertex v1
   Repeat
      Move to outgoing edges
      Increment the sack value by the edge weight
      Move from edges to incoming vertices
      Remove the path if it is a cycle
      Create a map; the keys are vertices, value is the minimum distance
01    Remove a traverser if its path is longer than the min path to the current v
02    Remove a traverser if it walked into a supernode with 100+ outgoing edges
03    Remove a traverser if its distance is greater than a global heuristic
   Check if the path reached v2
   Sort the paths by their total distance value
   Allow the first x paths to continue
   Shape the result
```

Let's walk through the process we described in Example 9-1 because we will be implementing it in Gremlin in this chapter. Our approach starts every traverser with a distance of 0.0 on the starting vertex. Then a looping condition moves a traverser onto an edge, updates the traverser's total distance, and applies a series of filters to determine whether the traverser should continue exploring the graph. We continue that

looping process until we find x number of paths that satisfy all of the optimizations and filters.

The series of filters we are referring to are labeled with O_n_ in Example 9-1 and apply each of the three optimizations we just outlined. The line labeled `O1` in Example 9-1 shows how we will apply the *lowest cost optimization*; a traverser will be removed if we have already found a shorter path to its location. The line labeled `O2` applies a global heuristic that removes a traverser if its path reaches too high of a weight, because such paths (probably) do not mean anything to your application. Last, the *supernode avoidance* optimization, labeled `O3`, filters out supernodes from our pathfinding algorithm by setting a hard limit on a vertex's degree.

We are almost ready to build up the Gremlin statements that implement Example 9-1. To help get us there, let's talk about how to address the edge weight problem from the end of Chapter 8.

# Normalization of Edge Weights for Shortest Path Problems

The way the dataset quantifies trust was the biggest hurdle we found during Chapter 8. With the way the data is now, we do not have a way to use the edge weights to find shortest weighted paths because the most trusted paths would be the longest ones.

We need to transform the edge weights to use them to find shortest weighted paths.

The upcoming transformation does two things. First, it applies logarithms so that we can meaningfully add weights to find maximum trust paths. Second, we have to flip the scale so that a minimum weighted path correlates to maximum trust.

This section walks through how to do this transformation. Then we will update our dataset and graph. Last, we will look at a few paths in the data and show how to meaningfully interpret the new edge weights.

## Normalizing the Edge Weights

There are three steps to the data transformation process:

1. Shift the scale to the interval [0,1].
2. Frame the new scale as a shortest path problem.
3. Decide how to handle modeling infinity.

Let's walk through all three of these steps and why we need to do them. We will show you how the scale transforms the weights at the very end.

### Step 1: Shift the scale to the interval [0,1]

The trust interval in the original dataset ranges from –10 to 10, where –10 represents no trust and 10 represents absolute trust. Figure 9-2 shows the distribution of observations in the dataset using Gremlin in DataStax Studio.

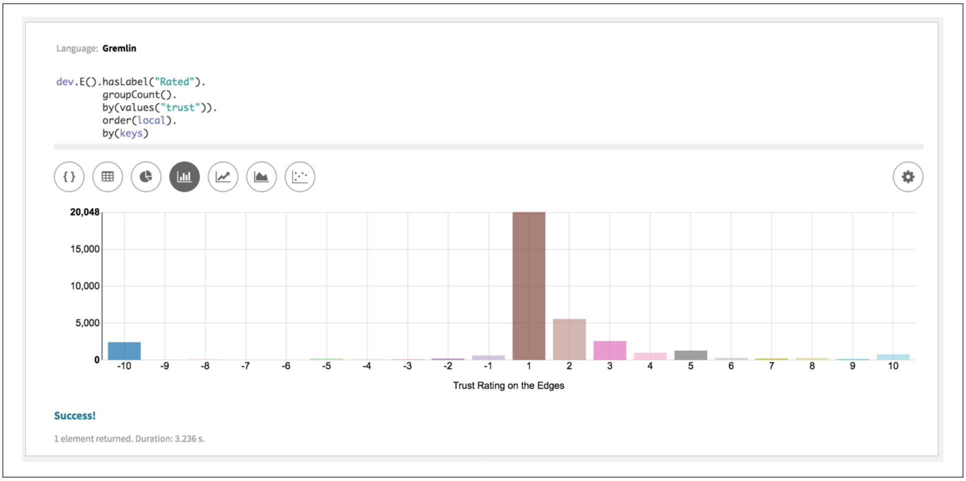



*Figure 9-2. The total number of observations for each trust rating in the Bitcoin OTC dataset*

Our objective is to map the trust scores from the interval [-10,10] onto the scale [0,1]. Mapping onto [0,1] gives us a way to create a confidence type score such that multiplying two scores gives us a mathematically sound way to model the aggregation of trust. This technique is similar to how we mathematically reason about probabilities.

In other words, mixing negative and positive scores together doesn't describe how we can mathematically reason about user ratings; we need a more consistent scale.

Additionally, we see in Figure 9-2 that there are no ratings with a trust value of 0. Therefore, we have decided that the rating of 1 will designate being "on the fence." And, we will remove "0" from the mapping. This mapping gives us the following starting points for our shifted scale:

1. A rating of –10 maps to 0 to mean no trust.
2. 1 maps to 0.5 to mean "on the fence."
3. 10 maps to 1 to mean maximum trust.

We will fill in the rest of the ratings linearly into those intervals. The linear transformation creates increments of `0.05` between –10 and 1 and increments of `0.05556` from 2 to 10. We calculated these increments via:

```
range/total_numbers = 0.5/10 = 0.05,
                    = 0.5/9  = 0.05556
```

The full table of mappings is coming up in Figure 9-4.

We can't yet use the values between 0 and 1 to calculate shortest paths on the edges because higher scores still correlate to high trust. We will run into the same problem as in Chapter 8: longer paths have higher trust. To get to where we need to be, we have to discuss two more mathematical transformations.

#### Step 2: Frame the new scale as a shortest path problem

We are essentially trying to find the *highest trust path* between two addresses in our data. To frame that as a shortest path problem, we have to do two things:

1. Use logarithms so that multiplication becomes addition.
2. Multiply the result by `-1`, so that the maximum becomes a minimum.

The first step here is an important transformation to understand so that you can accurately model certain phenomena in data, such as trust, so let's walk through it.

In many cases, using logarithms for edge weights isn't necessary because you can simply add up the weights, as in the logistics example.

However, in some cases, you need to multiply instead of add. This is true when you are dealing with probabilities, confidence values, and so on.

Trust is essentially a confidence value. Mathematically, this means that "your trust of someone else's trust" multiplies those two concepts, rather than adding them together.

Let's think about it. If you *half* trust person A, and person A *half* trusts person B, do you conclude that you fully trust person B? No, you probably don't. Instead, you conclude that you somewhat distrust person B.

How you are reasoning about this is the difference between adding trust scores and multiplying them. If you decided that you fully trusted person B, you would be adding half of your trust and half of person A's trust to reach your conclusion. Logically, this doesn't make sense, because we are dealing with your confidence in someone else's opinion. When you reason that you somewhat distrust person B, you are (essentially) multiplying your half trust by person A's half trust to arrive somewhere around "0.25" total trust.

To represent this numerically, we have to apply a logarithmic transformation to use the values between 0 and 1. Using logarithms allows us to add the trust scores together instead of multiplying them. This transformation gives us the following values for our scores:

1. –10 maps to 0; `log(0) = negative infinity`

2. 1 maps to 0.5; `log(0.5) = -0.301`
3. 10 maps to 1; `log(1) = 0`

The second half of step 2 indicates that the final transformation is to multiply these scores by `-1`. This last step is required so that the maximum becomes a minimum; we need minimums for finding shortest weighted paths.

To illustrate this mapping, Figure 9-3 plots our transformation. On the y-axis, 0 means distrust and 1 means trust. The x-axis shows how higher trust scores correlate to lower trust.

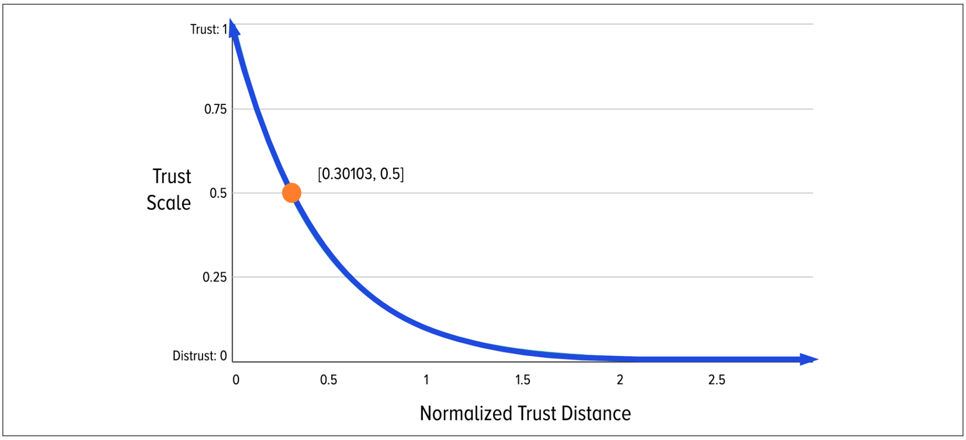



*Figure 9-3. Observing how a path's trust distance converts to trust or distrust on our shifted scale*

The point to consider is the point shown in Figure 9-3.

The point at which a trust score flips from trust to distrust is `0.30103`. A score of less than `0.30103` will represent trust, whereas value greater than `0.30103` represents distrust.

The transformation of high trust to low scores gives us the ability to add scores together such that lower total scores mean higher total trust. Being able to find lowest scores gives us an optimization to find the smallest total weight. From here, we can apply these new weights to reason about shortest weighted paths in our application.

There is one last decision required: how to represent `(-1)*log(0) = infinity` in our data.

### Step 3: Decide how to handle modeling infinity

There are a few decisions your team has to weigh about how to represent `(-1)*log(0) = infinity` in your data. You want to select a value large enough so that a

path with this value has little chance of being a shorter weighted path, but not so large that its value is worse than no edge at all.

We selected the value `100` to represent the score of `(-1)*log(0)`. Let's think about why this is a decent choice. Consider arbitrary endpoint vertices `a` and `b` with an edge weight of 100. The weighted path between `a` and `b` is almost guaranteed to be longer than any other path in the graph. You would have to find a path of 101 edges, with each edge having a weight of 1, for the direct path between `a` and `b` to be a shorter weighted path. In the context of our problem, a path of length 101 doesn't really make sense to our application. As a result, we feel the choice of `100` for our example is good enough.

The values shown in Figure 9-4 detail the steps we just discussed in the past few sections. We first shifted [–10, 10] to [0,1]. Then, we took the logarithm of each value and multiplied the result by `-1`. The final scores set `(-1)*log(0)` to 100.

| Trust | Shifted | Log(Shifted) | Final Values for norm_trust |
|---|---|---|---|
| -10 | 0 | negative infinity | 100 |
| -9 | 0.05 | -1.301 | 1.301 |
| -8 | 0.1 | -1 | 1 |
| -7 | 0.15 | -0.8239 | 0.8239 |
| -6 | 0.2 | -0.699 | 0.699 |
| -5 | 0.25 | -0.6021 | 0.6021 |
| -4 | 0.3 | -0.5229 | 0.5229 |
| -3 | 0.35 | -0.4559 | 0.4559 |
| -2 | 0.4 | -0.3979 | 0.3979 |
| -1 | 0.45 | -0.3468 | 0.3468 |
| 1 | 0.5 | -0.301 | 0.301 |
| 2 | 0.5556 | -0.2553 | 0.2553 |
| 3 | 0.6111 | -0.2139 | 0.2139 |
| 4 | 0.6667 | -0.1761 | 0.1761 |
| 5 | 0.7222 | -0.1413 | 0.1413 |
| 6 | 0.7778 | -0.1091 | 0.1091 |
| 7 | 0.8333 | -0.0792 | 0.0792 |
| 8 | 0.8889 | -0.0512 | 0.0512 |
| 9 | 0.9444 | -0.0248 | 0.0248 |
| 10 | 1 | 0 | 0 |

*Figure 9-4. The full table of values to transform weights from –10 to 10 to values that can be used for finding shortest weighted paths*

Next, we need to update our graph's schema and load the transformed version of the edges so that we can use these new weights.

## Updating Our Graph

We want to augment our current `rated` edge to have the new normalized values. We will do that by adding a property called `norm_trust` onto the `rated` edge as the clustering key. Figure 9-5 shows the new graph model and indicates that we are making the new property the clustering key for the edges. Indicating that `norm_trust` is an edge's clustering key will sort the `rated` edges on disk in increasing order.

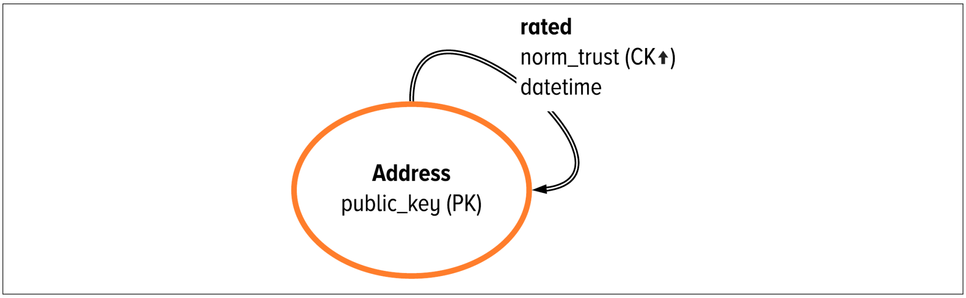



*Figure 9-5. Using GSL notation, the production schema for our graph model*

The schema code for Figure 9-5 is in Example 9-2. We hope you are learning how to translate graph data models to schema code with the Graph Schema Language (GSL), just like using an ERD to create tables.

*Example 9-2.*

```
schema.vertexLabel("Address").
       ifNotExists().
       partitionBy("public_key", Text).
       create();

schema.edgeLabel("rated").
       ifNotExists().
       from("Address").
       to("Address").
       clusterBy("norm_trust", Double, Asc).
       property("datetime", Text).
       create()
```

As we did in Chapter 8, we are going to load the data into our graph using the DataStax Bulk Loader, a command-line tool. The datasets that accompany this text already have the transformation of the edge weights. If you would like to see the code, please head to the Chapter 9 data directory within this book's GitHub repository for the data and loading scripts for these examples.

Let's do some basic exploratory queries to ensure that we understand our data and that it loaded correctly.

## Exploring the Normalized Edge Weights

Before we get into implementing shortest weighted paths, let's look at our same queries from Chapter 8. This time, however, we want to use the `norm_trust` property as we explore paths between `1094` and `1337`.

The two queries we will do in this section are:

1. Find all paths of length 2, sorted by total trust
2. Find the 15 shortest paths by path length, sorted by total trust

Let's start with the first query.

### Find all paths of length 2, sorted by total trust

In Example 9-3, we are revisiting the same path of length 2 from Chapter 8 but are calculating the trust distance using the normalized weights.

*Example 9-3.*

```
g.withSack(0.0).
  V().has("Address", "public_key", "1094").
  repeat(outE("rated").
         sack(sum).
           by("norm_trust").
         inV()).
  times(2).
  has("Address", "public_key", "1337").
  order().
    by(sack(), asc).
  project("path_information", "total_elements", "trust_distance").
    by(path().by("public_key").by("norm_trust")).
    by(path().count(local)).
    by(sack())
```

The raw results of Example 9-3 are shown in Example 9-4:

*Example 9-4.*

```
{"path_information": {
    "labels":  [[],[],[],[],[]],
    "objects": ["1094", "0.0248", "1268", "0.30103", "1337"]
 },
 "total_elements": "5",
 "trust_distance": "0.32583"
}
```

As we found during Chapter 8, there is only one path of length 2 between our start and end vertices. The path object from Example 9-4, combined with the weights we found in Chapter 8, is illustrated in Figure 9-6.

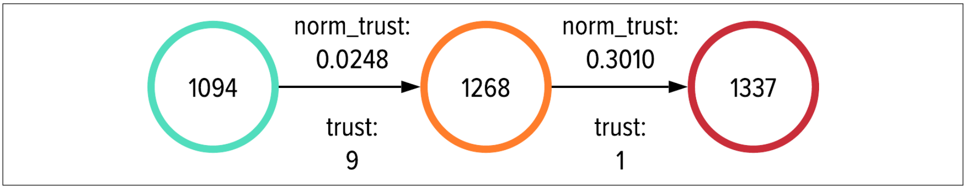



*Figure 9-6. Observing the normalized edge weights on the only path of length 2 in the data between `1094` and `1337`*

The total trust distance for the path illustrated in Figure 9-6 is 0.32583. You can reverse this score to understand how it would fit into the shifted [0, 1] scale. To do that, you multiply the final score by `-1` and then raise 10 to the power of the result: `10^(-1*(0.0248 + 0.3010)) = 0.4723`.

This means that the weighted trust of this path on a scale of [0,1] is `0.4723`. Thus, we *slightly distrust* this path because 0 means distrust and 1 means trust. This path's weighted trust score is slightly less than `0.5` and is therefore slightly distrusted.

You may be wondering: but what about other paths? So, let's look at our second query from Chapter 8.

### Find the 15 shortest paths by path length, sorted by total trust

For a quick refresher, remember that the queries we built up in Chapter 8 combined our knowledge of barriers in Gremlin and the logic of breadth-first search. The queries applied these concepts to guaranteed shortest paths by path length, not by weight.

We apply the shortest path logic in Example 9-5 to find the 15 shortest paths by length, but then order those paths by their normalized trust distance.

Let's see the query in Example 9-5.

*Example 9-5.*

```
g.withSack(0.0).           // init each traverser to have a value of 0.0
      V().has("Address", "public_key", "1094").     // start at 1094
      repeat(              // repeat
        outE("rated").     // walk out to an edge and stop
         sack(sum).        // aggregate into the traverser's sack
           by("norm_trust").// the value on the edge's property: "norm_trust"
         inV().            // move and walk into the next vertex
        simplePath()).     // remove the traverser if it has a cycle
      until(has("Address", "public_key", "1337")).   // until you reach 1337
      limit(15).           // BFS: first 15 are the 15 shortest paths, by length
      order().             // sort the 15 paths
        by(sack(), asc).   // by their aggregated trust scores
      project("path_information", "total_elements", "trust_distance"). // make a map
        by(path().by("public_key").by("norm_trust")).// first value: path information
        by(path().count(local)).                     // second value: length
        by(sack())                                   // third value: trust
```

Example 9-6 displays the results of Example 9-5, and our top three most trusted paths show very interesting results. In Chapter 8, we found the shortest path: `1094 → 1268 → 1337`. Example 9-6 shows that this path is the second most trusted path of the 15 shortest paths, which means we can conclude that there is a longer path that is also more trusted.

*Example 9-6.*

```
{
  "path_information": {
    "labels": [[],[],[],[],[],[],[]],
    "objects": ["1094","0.2139","280","0.0512","35","0.0248","1337"]
  },
  "total_elements": "7",
  "trust_distance": "0.2899"
},...,
{
  "path_information": {
    "labels":  [[],[],[],[],[]],
    "objects": ["1094","0.0248","1268","0.30103","1337"]
  },
  "total_elements": "5",
  "trust_distance": "0.32583"
},
{
  "path_information": {
    "labels":  [[],[],[],[],[],[],[]],
    "objects": ["1094","0.0248","1268","0.30103","35","0.0248","1337"]
    },
```

```
  "total_elements": "7",
  "trust_distance": "0.35063"
},...
```

Example 9-6 displays results that we couldn't find in Chapter 8: a path that is longer and more trusted. The most trusted of the 15 shortest paths is a path of length 3, `1094 → 280 → 35 → 1337`, with a total trust score of `0.2899`.

> The results in Example 9-6 are the shortest paths by length, sorted by their trust distance. This is not the same as shortest weighted paths, which we have not done yet.

It is exciting to have found a longer path with a better trust score. However, what does the value `0.2899` mean? How much do we trust this path?

#### Interpreting path distance to total trust with the normalized edge weights

The weights in our graph represent a normalized trust distance. This guarantees that the shortest weighted path is the most trusted path.

Ultimately, you want to say, "Do I trust this path or not?" To answer that, you have to convert the path's final weight back to the shifted scale from Figure 9-4. You have to convert the path's total trust distance to make a statement about whether you trust or distrust this path.

Let's deeply examine how a path's trust distance maps back to our trust scale of [0,1].

The best possible shortest path has a weight of zero; this would happen when all edges in the path have a normalized weight of 0. A path's trust distance of 0 converts to that path having the highest trust score of 1: `10^(-0)=1`.

For all trust distances `d`, the conversion formula is shown in Figure 9-7 and Figure 9-3.

$$f\left(d\right) = 10^{(-d)}$$

*Figure 9-7. For a normalized trust distance d, the formula for converting a path's distance to the trust scale of [0,1]*

For all three results from Example 9-6, let's convert their total weight. Each path and their conversion is:

1. Top path with seven objects: 10^(–0.28990) = 0.5130
2. Shortest path with five objects: 10^(–0.32583) = 0.4722
3. Third path with seven objects: 10^(–0.35063) = 0.4460

The three converted scores above represent each path's total trust on the scale [0,1], where 0 means distrust and 1 means trust. This means that of our 15 shortest paths by length, we found one path that we slightly trust. The top result from Example 9-6 has an aggregated normalized weight of `0.28990`, which converts to a trust score of `0.5130` on our +[0,1] scale. Therefore, we slightly trust this path.

The past examples helped us understand how we were going to reason about the normalized trust scores in our paths.

However, is the first result from Example 9-6 the most trusted path in our data? To find out, we need to apply some optimizations to our query to find the single shortest weighted path.

## Some Thoughts Before Moving On to Shortest Weighted Path Queries

The data we are using for this example aims to show you how to find the most trusted path between two addresses. The most trusted paths in this data are more than just the shortest paths.

We want to find the paths through our example with the highest trust values from their edges.

To get there, we had to convert the edge weights so that they can be used to solve the shortest weighted path problem. The conversion process did two things: (1) it used logarithms so that we can meaningfully add weights along the path, and (2) it flipped the scale so that a minimum weighted path correlates to maximum trust.

We are iterating this process because these are common tools that teams use so they can use weighted edges in shortest path problems. Reshaping data to solve complex problems illustrates the powerful creativity within the intersection of data science and graph applications.

Using this knowledge, let's move on to developing Gremlin queries that calculate shortest weighted paths.

# Shortest Weighted Path Queries

The algorithmic process we have been using up to this point is shown in Example 9-7.

*Example 9-7.*

```
A  Use sack to initialize the path distance to 0.0
B  Find your starting vertex v1
C  Repeat
D      Move to outgoing edges
E      Increment the sack value by the edge weight
F      Move from edges to incoming vertices
G      Remove the path if it is a cycle
H  Check if the path reached v2
I  Allow the first 20 paths to continue
J  Sort the paths by their total distance value
K  Shape the result
```

Recall that barrier steps in Gremlin, like the `repeat().until()` pattern, process the data like breadth-first search. This means that step `I` in Example 9-7 guarantees shortest paths by length.

In Example 9-8, those algorithmic steps are shown next to the corresponding lines in the query we just did.

*Example 9-8.*

```
A  g.withSack(0.0).
B    V().has("Address", "public_key", "1094").
C    repeat(
D           outE("rated").
E           sack(sum).by("norm_trust").
F           inV().
G           simplePath()).
H    until(has("Address", "public_key", "1337")).
I    limit(20).
J    order().
       by(sack(), asc).
K    project("path_information", "total_elements", "trust_distance").
       by(path().by("public_key").by("norm_trust")).
       by(path().count(local)).
       by(sack())
```

We are going to use the pattern of pseudocode, as in Example 9-7, and mapping the process to Gremlin steps, as in Example 9-8, to build up our shortest weighted path query. We will add to the query in Example 9-8 to change and then optimize it to be a shortest weighted path query.

## Building a Shortest Weighted Path Query for Production

The process we want to build toward implements the optimizations we introduced in "Shortest Weighted Path Search Optimizations" on page 264: lowest cost optimiza-

tion, global heuristics, and supernode avoidance. There are four steps to doing this so that we can create a production-quality query for our application:

1. Swap two steps and change our limit
2. Add an object to track the shortest weighted path to a visited vertex
3. Remove a traverser if its path is longer than one already discovered to that vertex
4. Remove traversers for custom reasons, such as to avoid supernodes

Let's build up the Gremlin query by incrementally adding steps through each of these four procedures.

#### 1) Swap two steps and change our limit

We went through a reminder of where we are because the first step in building our shortest path query is very similar to Example 9-8. We need only to swap the order of steps and limit to one result to translate Example 9-8 to a single shortest weighted path query.

The algorithm in Example 9-9 swaps step `J` and step `I` from Example 9-7. This swap changes our process from shortest paths to shortest weighted paths. Then we change the limit from 20 to 1 so that we are finding the single shortest weighted path.

We are going to start labeling these new optimizations with `O_stepNumber` and their step number from this section. You will find asterisks (*) in the pseudocode and query to indicate the new lines we are adding to our pathfinding traversal. We find this easier for logically mapping the optimization in the pseudocode to the statements in the Gremlin query.

*Example 9-9.*

```
A   Use sack to initialize the path distance to 0.0
B   Find your starting vertex v1
C   Repeat
D       Move to outgoing edges
E       Increment the sack value by the edge weight
F       Move from edges to incoming vertices
G       Remove the path if it is a cycle
H   Check if the path reached v2
O1* Sort the paths by their total distance value
O1* Allow the first path to continue, this is the shortest path by weight
K   Shape the result
```

If it is that easy, why can't we just stop here?

We can, but, well…there's a but.

Swapping the steps in Gremlin introduces another barrier step, `order()`. The presence of `order()` immediately after `repeat().until()` means that we have to find and sort *all paths*, not just the shortest weighted paths. So we will need to add a bit more to the query to optimize it.

The swapping of these steps, however, does guarantee that the paths that we shape at `step K` are ordered according to their total distance. This is ultimately what we want; we are just processing more data than we want because we are still finding all paths.

Let's see where we will be starting our query building in Example 9-10 with the swapped logic from Example 9-9. The changes we built are labeled with `01*`, to indicate this is the first optimization we have built so far.

*Example 9-10.*

```
A   g.withSack(0.0).
B     V().has("Address", "public_key", "1094").
C     repeat(
D,E,F      outE("rated").sack(sum).by("norm_trust").inV().
G          simplePath()
H     ).until(has("Address", "public_key", "1337")).
01*   order().
        by(sack(), asc).
01*   limit(1).
K     project("path_information", "total_elements", "trust_distance").
        by(path().by("public_key").by("norm_trust")).
        by(path().count(local)).
        by(sack())
```

Example 9-10 finds all weighted paths between `1094` and `1337`. You do not want to use this yet in a production application because finding all paths is too computationally expensive. There are multiple optimizations we can apply to ensure that we create a query that is safer to run in a distributed graph in production.

### 2) Add an object to track the shortest weighted path to a visited vertex

The construction of an object to track shortest weighted paths will be used many times in the coming optimizations.

The idea is to create a map. The keys of the map will be visited vertices, and the value will track the shortest distance to that vertex. The additional processes to the algorithm are shown in Example 9-11. We map the procedures from Example 9-11 to the Gremlin query in Example 9-12.

*Example 9-11.*

```
A    Use sack to initialize the path distance to 0.0
B    Find your starting vertex v1
C    Repeat
D       Move to outgoing edges
E       Increment the sack value by the edge weight
F       Move from edges to incoming vertices
G       Remove the path if it is a cycle
O2*     Create a map; the keys are vertices, value is the minimum distance
H    Check if the path reached v2
O1   Sort the paths by their total distance value
O1   Allow the first x paths to continue
K    Shape the result
```

The query that applies the algorithm from Example 9-11 is Example 9-12. The changes we built are labeled with `O2*`, to indicate this is the second optimization we have built so far.

*Example 9-12.*

```
A  g.withSack(0.0).
B     V().has("Address", "public_key", "1094").
C     repeat(
D,E,F      outE("rated").sack(sum).by("norm_trust").inV().
G          simplePath().
O2*        group("minDist").    // create a map
O2*           by().             // the keys are vertices
O2*           by(sack().min())  // the values are the min distance
H     ).until(has("Address", "public_key", "1337")).
O1    order().
O1    by(sack(), asc).
O1    limit(1).
K     project("path_information", "total_elements", "trust_distance").
K       by(path().by("public_key").by("norm_trust")).
K       by(path().count(local)).
K       by(sack())
```

Let's talk about the map we constructed on the lines labeled `O2*` in Example 9-12. This map contains keys and values where the keys are vertices. The trick here is in how we set up the values: `by(sack().min())`. The values in this map will be the *minimum* distance to any visited vertex in the graph.

Essentially, this map creates a lookup table that every traverser can access and ask: what is the current minimum distance to my current vertex?

Now that we have created this map, let's use it.

### 3) Remove a traverser if its path is longer than one already discovered to that vertex

The map `minDist` tracks a visited vertex and the minimum distance to that vertex. Let's use this map.

For any traverser in our stream, we want to do two things. First, we want to use the map to look up the minimum distance we have seen so far to that vertex. Then we want to compare that value to the current traverser's traveled distance.

If the distances are the same, that means the current traverser is on the shortest path to the current vertex. If the traverser's distance is greater than the shortest distance, then it is exploring a longer weighted path, and we want to remove it from the traversal pipeline. There will not be a case when the traverser's distance is less because we update the map before we do this comparison.

Let's look at the pseudocode that describes this process. The new optimization is labeled with `O3*` in Example 9-13.

*Example 9-13.*

```
A   Use sack to initialize the path distance to 0.0
B   Find your starting vertex v1
C   Repeat
D      Move to outgoing edges
E      Increment the sack value by the edge weight
F      Move from edges to incoming vertices
G      Remove the path if it is a cycle
O2     Create a map; the keys are vertices, value is the minimum distance
O3*    Remove a traverser if its path is longer than the min path
H   Check if the path reached v2
O1  Sort the paths by their total distance value
O1  Allow the first x paths to continue
K   Shape the result
```

To implement Example 9-13 in Gremlin, we will need to introduce two new patterns of steps. First, we will create a custom filter with the `filter()` step.

*filter()*
: The `filter()` step evaluates the traverser to either true or false, where false will not pass the traverser to the next step.

Inside the filter step, we will use a new pattern. We need to create a pattern that evaluates two values, `a` and `b`. The common way to do this in Gremlin is to create a map and then test the objects in the map with the `where()` step.

*where()*
: The `where()` step filters the current object; in our examples, we will filter based on the object itself.

*project().where()*

The `project().where()` pattern tests the objects in the map according to a provided condition in the `where()` step.

Let's see these steps in action in Example 9-14.

*Example 9-14.*

```
A g.withSack(0.0).
B     V().has("Address", "public_key", "1094").
C     repeat(
D,E,F        outE("rated").sack(sum).by("norm_trust").inV().as("visited").
G            simplePath().
02           group("minDist").
02             by().
02             by(sack().min()).
03*          filter(project("a","b").                        // boolean test
03*                   by(select("minDist").select(select("visited"))). // a
03*                   by(sack()).                                     // b
03*                 where("a",eq("b"))                       // does a == b?
H     ).until(has("Address", "public_key", "1337")).
01     order().
01       by(sack(), asc).
01     limit(1).
K     project("path_object", "total_elements", "trust_distance").
K       by(path().by("public_key").by("norm_trust")).
K       by(path().count(local)).
K       by(sack())
```

Let's describe what is happening with our new optimization lines, labeled `03*`. We create a boolean test for a traverser with the `filter()` step: if the condition is true, the traverser will survive. The test uses the `project().where()` pattern to set up and compare variables: `a` and `b`. The value for `a` uses our map `minDist` to get the minimum distance to the current vertex. Then we look up the traverser's current sack value; this is the value for `b`.

If the current minimum distance to the vertex is equal to the traverser's sack, the test resolves to `True` and the traverser survives. This means that the traverser is on the shortest path, so we want it to continue exploring the graph.

If you have been following along in the notebook, Example 9-14 is the first time our weighted path queries are able to return without a `timeOut` error. This is because this optimization is the first step toward the reduction of paths that we process in the query. The first two optimizations were setting us up to apply them at the lines labeled `03*` in Example 9-14.

There are a few more ways that we can prune paths from our working tree.

#### 4) Remove traversers for custom reasons, such as to avoid supernodes

A common optimization we add reduces the search space to address the computational complexity of path queries. Specifically, we want to filter out a traverser from the pipeline if it has arrived at a supernode. The definition of a supernode will vary according to your dataset, like the celebrity problem within the Twitter graph that we talked about in "Supernodes in graphs" on page 265.

Let's take a look at this graph's degree distribution, shown in Figure 9-8.

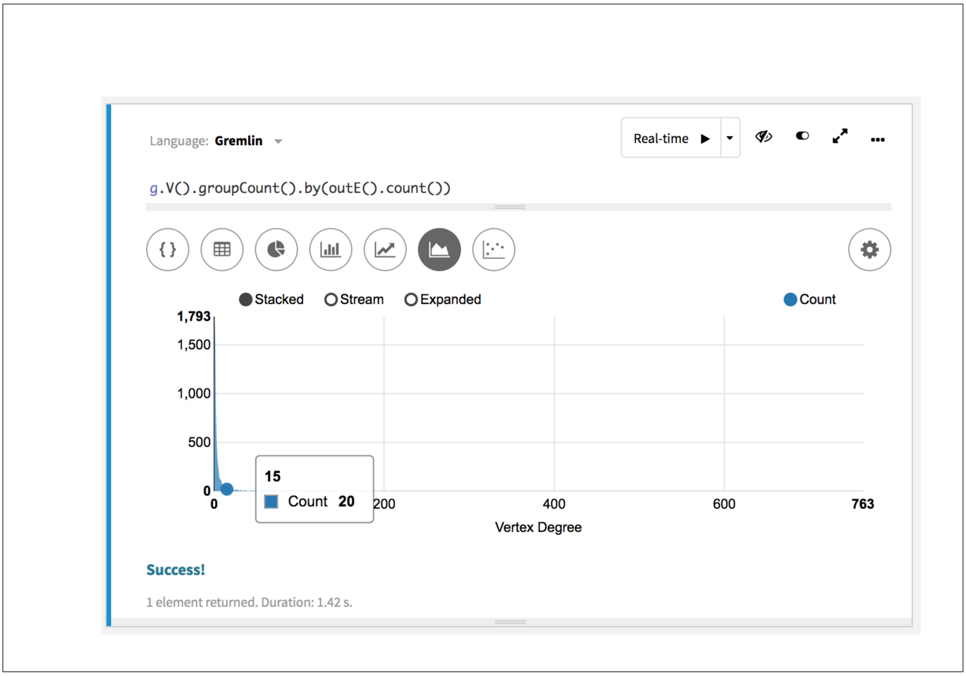



*Figure 9-8. The degree distribution for the graph used in this example*

The outgoing degree distribution of this graph shows that most vertices have, say, 20 or fewer outgoing edges. The far right value in Figure 9-8 shows that the outlier in our dataset has 763 outgoing edges.

For illustrative purposes, let's say that we want to exclude vertices with 100 or more outgoing edges. The pseudocode in Example 9-15 shows where we will apply this filter with the label `04*`. Example 9-16 shows the Gremlin query.

*Example 9-15.*

```
A   Use sack to initialize the path distance to 0.0
B   Find your starting vertex v1
C   Repeat
```

```
D       Move to outgoing edges
E       Increment the sack value by the edge weight
F       Move from edges to incoming vertices
G       Remove the path if it is a cycle
02      Create a map; the keys are vertices, value is the minimum distance
03      Remove a traverser if its path is longer than the min path to the current v
04*     Remove a traverser if it walked into a supernode; 100 outgoing edges or more
04*     Remove a traverser if its distance is greater than what we want to process
H    Check if the path reached v2
01   Sort the paths by their total distance value
01   Allow the first x paths to continue
K    Shape the result
```

The query that implements Example 9-15 is shown in Example 9-16.

*Example 9-16.*

```
max_outgoing_edges = 100;
max_allowed_weight = 1.0;

A g.withSack(0.0).
B      V().has("Address", "public_key", "1094").
C      repeat(
D,E,F         outE("rated").sack(sum).by("norm_trust").inV().as("visited").
G             simplePath().
02            group("minDist").
02              by().
02              by(sack().min()).
03            and(project("a","b").
03                 by(select("minDist").select(select("visited"))).
03                 by(sack()).
03                where("a",eq("b")),
04*               filter(sideEffect(outE("rated").count().// optimization:
04*                      is(gt(max_outgoing_edges)))),    // remove supernodes
04*               filter(sack().                          // optimization:
04*                      is(lt(max_allowed_weight))))     // global heuristic
H      ).until(has("Address", "public_key", "1337")).
01      order().
01        by(sack(), asc).
01     limit(1).
K      project("path_object", "total_elements", "trust_distance").
K        by(path().by("public_key").by("norm_trust")).
K        by(path().count(local)).
K        by(sack())
```

Let's walk through the new steps labeled `04*` in Example 9-16. We added two boolean tests. The first one is a `filter()` that checks for the current vertex's outgoing degree and compares it to our supernode threshold. If the degree is higher than 100, the traverser fails the test and is removed from the traversal pipeline. This is how you can specifically remove supernodes from your pathfinding query.

The supernode avoidance optimization required us to use `sideEffect()`; we will explain why in the next section.

The second optimization in Example 9-16 adds another `filter()` and applies a global heuristic. We set the maximum weight we want to consider to 1.0, and we test for this by comparing the traverser's `sack()` to our threshold. If the traverser's distance is greater than 1.0, it will fail the test and be removed from the pipeline.

The query in Example 9-16 wrapped all of our optimizations within a new step, `and()`. We will explain `and()` and `sideEffect()` in the next two sections and then we will look at the results from Example 9-16.

**The and() step in Gremlin.** The `and()` step in Gremlin is a filter that can have an arbitrary number of traversals. The `and()` step applies a boolean AND to the results from each traversal to create a pass/fail condition for the traverser.

*and()*
: The `and(t1, t2, …)` step in Gremlin yields true or false for each traverser in the pipeline according to the values for each input traversal `t1`, `t2`, and so on.

In the context of Gremlin, each traversal within the `and()` step must produce at least one output. In the context of boolean operations, the following values are interpreted as false:

1. `False`
2. Numeric zero of all types
3. Empty strings
4. Empty containers (including tuples, lists, dictionaries, sets, and frozen sets)

All other values are interpreted as `True`.

There are three traversals that are wrapped within the `and()` step in Example 9-16. Each traversal's boolean condition is tested, and all three results are analyzed with `and()`. Only if all three conditions yield true will the traverser pass. This means that the traverser must be on a shortest path, not on a supernode, and have a total distance less than 1.0.

The last main concept to understand is why we needed to use `sideEffect()` for our supernode test.

**sideEffect() in Gremlin.** One of the most valuable heuristics you can apply to a path-finding query removes traversers when they are on a supernode. You have to count the edges of the current vertex to figure out whether or not it is a supernode. When you are on a vertex, you have to move to all outgoing edges to count them.

Moving from the current vertex to all outgoing edges changes the location of the traverser. When we are in the middle of a pathfinding query, this would change the location of our traverser from a vertex to a set of edges. This change would break the conditional flow of our `repeat()` step.

Therefore, we have to check whether the current vertex is a supernode and do so in a way that doesn't change the state of the current traverser (or doesn't move the traverser). We can do these types of side computations by using `sideEffect()`, one of the five general ways that a traverser can move throughout a graph.

*sideEffect()*
: The `sideEffect(<traversal>)` step allows the traverser's state to proceed unchanged to the next step, but with some computational value from the provided traversal.

We used `sideEffect(outE("rated").count().is(gt(max_outgoing_edges)))` in Example 9-16. Let's break this down.

First, the traversal that is wrapped within `sideEffect()` is `outE("rated").count().is(gt(max_outgoing_edges))`. This asks the question: is the number of outgoing edges on this vertex less than the maximum we are allowing?

To answer that question, the traverser has to move from the vertex to all outgoing edges, count them, and compare the result to `max_outgoing_edges`. The problem is that we have to move. We do not want this move to affect the state of the traversal, so we wrap this traversal in `sideEffect()` so that whatever we do in this embedded traversal doesn't change where the traverser is located in the graph as it moves to the step after `sideEffect(<traversal>)`.

This gives us everything we need to know about how the traversal works. Let's look at and interpret the results of Example 9-16.

**Interpreting the results of our shortest weighted path.** The last set of results to understand for our chapter's examples is shown in Example 9-17. Let's look at the shortest weighted path now.

*Example 9-17.*

```
{
  "path_information": {
      "labels": [<omitted in text>],
      "objects": ["1094",
                  "0.0","64",
                  "0.0","104",
                  "0.0","23",
                  "0.0792","1217",
                  "0.0248","1437",
```

```
                     "0.0","35",
                     "0.0248","1337"]
 },
  "total_elements": "15",
  "trust_distance": "0.1288"
}
```

Our final shortest weighted path had a total trust distance of `0.1288` and has a length of 7! (15 elements means eight vertices and seven edges; path length is the number of edges in the path.)

The trust distance is `0.1288`, where `10^(-0.1288) = 0.7434`, so we conclude that we trust this path.

Thinking back to the broader application, we also conclude that we trust accepting bitcoins from `1337`. What would you do?

## Weighted Paths and Trust in Production

Whether or not you actively trade Bitcoin, you've already integrated the concept of weighted paths and determining trust into your daily life. You may not go through transforming the data into a logarithmically normalized graph, but we would bet that you use the concepts from the past two chapters in some way.

The beauty of graph technology lies in translating natural human tendencies into quantifiable models. Throughout Chapter 8 and so far in this chapter, we walked through many different ways of translating natural thinking into metrics and models. We showed you how to use the idea of distance between people and concepts to teach you how we think about data to solve complex path problems in production.

The big moment here is that you naturally make decisions and inferences about previously disconnected topics. And this naturally occurring process you already do maps very well to graph technology to quantify your decision in a repeatable framework.

Graph technology gives us a framework for defining, modeling, quantifying, and applying mental processes that we take for granted, like correlating path distance to trust. This is what makes graph technology so beautiful and impactful. The things we already naturally do without thinking can be formally and logically defined with technology that represents them in the same way.

So how would you rate your trust in us given all the stops along your journey throughout this book? Once you start thinking about your journey, would you assign different strengths to different pieces of your journey?

Maybe you should consider using author and content ratings to create a graph of trustable resources, which seems oddly reminiscent to how Netflix ignited the

journey of graph thinking with movie recommendations based on user ratings. And that sounds like a great topic to visit next.

Our next chapter is going to be a Netflix-like example in which we show you how to recommend movies based on user ratings.

# CHAPTER 10
# Recommendations in Development

The Netflix Prize was an open machine learning competition started in 2006. Each team that entered the competition aimed to build an algorithm capable of besting Netflix's own content rating prediction process. The competition awarded $1 million to the winning team in 2009.

One specific derivative of the Netflix Prize sent waves throughout the graph theory community, a result you are experiencing now as you read this book. The competition ignited the use of graph thinking as a solution for traditionally matrix-based algorithms.

The realization was that it is much easier to explain recommendation systems with a graph than with a matrix representation. Think about it. You have a favorite set of movies, and each of those movies is highly rated by other people. If you look at the other movies liked by those people, you have a list of movies that you may also like. You have a list of movie recommendations.

And you just walked through a graph to find them.

The Netflix Prize[1] popularized the idea of using relationships between users and movies to predict and personalize your digital experience. This small idea of thinking about your data like a graph has become one of the main drivers of the rise of graph thinking.

We will bring this idea to life throughout this chapter and Chapter 12. And in case you are wondering, Chapter 11 shows you how we created the graph model you will see in this chapter.

1 James Bennett and Stan Lanning, "The Netflix Prize," _Proceedings of KDD Cup and Workshop, 2007.

## Chapter Preview: Collaborative Filtering for Movie Recommendations

In this chapter, we will show and define collaborative filtering by walking through how a site/app makes movie recommendations to its users.

In the first section, we are going to walk through three different examples of recommendation systems. These three examples illustrate how deeply ingrained the use of graph thinking has become for customizing a user's experience in an application. You likely use these techniques every day, perhaps without even knowing it.

The second section will walk through an introduction to collaborative filtering. We will focus on item-based collaborative filtering because it is the most popular way to use graph structures for recommendations.

The third section will introduce two open source datasets for our movie recommendations example. We will build a complex schema and show you the data structures and loading procedures. We will be using this data throughout the next two chapters.

Then we'll take a short side tour and use the complex data model for the movie datasets as a review of this book's main techniques. We will explore the merged datasets by revising the three most popular production queries with graph data: neighborhoods, trees, and paths.

The last section of this chapter steps through doing item-based collaborative filtering in Gremlin. As you have seen in the trees and paths chapters, we will run into a problem at the end of this chapter due to the scalability of doing collaborative filtering in real time.

# Recommendation System Examples

The popularity of recommendation systems with graphs derives from the simplicity of explaining how they work. Let's follow a deeper progression through graph structure, one neighborhood at a time, to show recommendations in three different industries.

We want to start our examples with how we see the problem now.

## How We Give Recommendations in Healthcare

If you think back through some of your most recent interactions with doctors, you probably have a short list of the doctors you trust the most.

Now, what would you say if your friend asked you to recommend a doctor?

To give a personal recommendation, you consider a plethora of factors, such as the outcome of your last visit, how you were treated, how expensive it was, and so on. Therefore, you likely didn't respond immediately to your friend's question with your favorite doctor; instead, you probably asked your friend for more information.

You need more context from your friend to make sure your recommendation is relevant to them. You use the additional details you gather about your friend's question to match them up to your experiences, and then you customize your recommendation.

How you ultimately respond and give a recommendation about healthcare probably looks something like the drawing in Figure 10-1. It is your response to your friend that shows how you think about recommendations like the first neighborhood of your personal health graph.

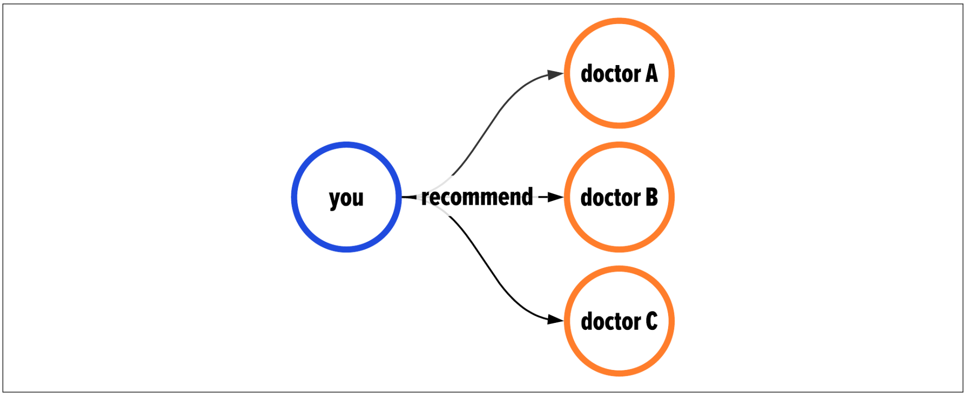



*Figure 10-1. An example of how we naturally think about the recommendations that we give*

It is the deeper details behind your recommendation that make it relevant.

For a much less personal topic than healthcare, let's see how deeper information from a graph structure can be used to create recommendations in your social media accounts.

## How We Experience Recommendations in Social Media

Think about the last time that you logged in to LinkedIn. Did you see a notification for "people you may know"?

The "people you may know" section is an example of using graph structure to recommend new connections on social media. This section also illustrates how to use your second neighborhood to create a recommendation. Figure 10-2 shows how to build up a list of people you may know in a graph.

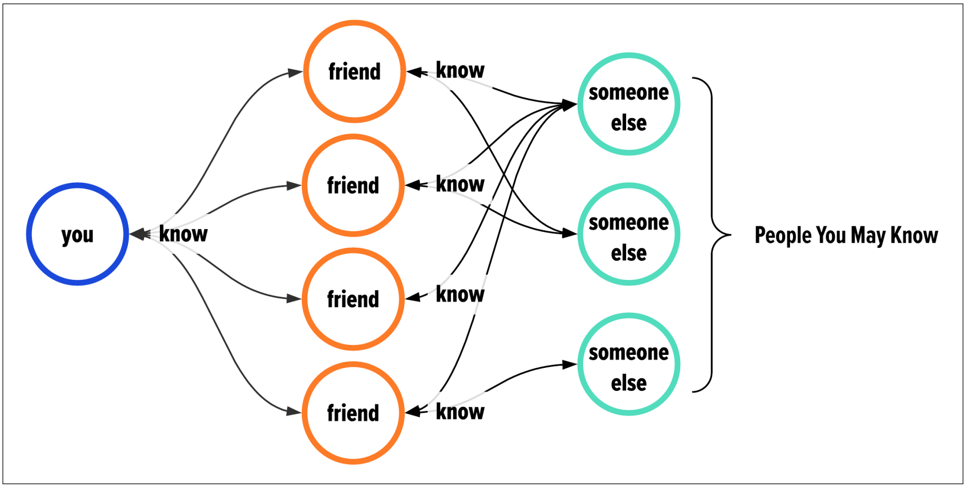



*Figure 10-2. An example of how we experience recommendations in social media*

Let's walk through how the concept from Figure 10-2 works on LinkedIn or on any other social media platform.

Over time, you have built up a list of friends on your social media account; Figure 10-2 illustrates this with the list of friends in orange. The friends of your friends form the list of people that you may also know, as shown further to the right in Figure 10-2.

It really is that simple. People you may know are in your second neighborhood of friends on social media.

This usually prompts the question: who is the person you are most likely to already know? And you probably already guessed the answer. The most connected friend of your friends is the top recommended person that you may also know.

The examples so far show shallow walks through first and second neighborhoods of graph data. Let's see a walk that goes a bit deeper.

## How We Use Deeply Connected Data for Recommendations in Ecommerce

The section of recommended products has become an expectation for any online retailer. People want to search for a product and then explore the company's catalog of similar products.

Product recommendations can be generated by walking through deeper neighborhoods of connected data. Figure 10-3 shows how a product you purchased can create a recommendation of three other products.

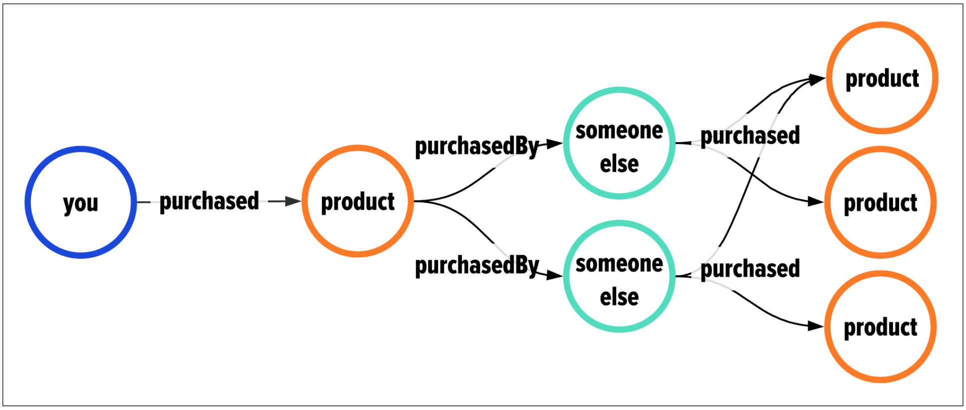



*Figure 10-3. An example of how we walk deeply through graph-structured data for recommendations in ecommerce*

Let's think about what Figure 10-3 is showing. It starts by showing one product that you purchased. Your online retailer also knows other people bought that product, as well as the other products they bought. The final list of three products on the far right in Figure 10-3 becomes the three products you see in a "similar products" window while you shop online.

Walking through Figure 10-3 also gives you your first glimpse into how collaborative filtering works. Let's delve into the algorithm we will be implementing in this chapter.

# An Introduction to Collaborative Filtering

Using collaborative filtering with graph-structured data is a proven technique for personalizing content recommendations. The industry defines *collaborative filtering* as follows:

*Collaborative filtering*
: Collaborative filtering is a type of recommendation system that predicts new content (filtering) by matching the interests of the individual user with the preferences of many users (collaborating).

Let's look at a quick introduction to the problem domain of recommendation systems and collaborative filtering.

## Understanding the Problem and Domain

Collaborative filtering is a very popular technique within the graph community. But it is better known for the larger role it plays within the class of recommender systems. Generally speaking, collaborative filtering is one of four types of automated algorithms that fall within the class of recommender systems. The other three are

content-based, social data mining, and hybrid models.footnote:[Loren Terveen and Will Hill, "Beyond Recommender Systems: Helping People Help Each Other." _HCI in the New Millennium_, ed. Jack Carroll (Boston: Addison-Wesley, 2001), 487–509.

To give you an idea of how all these concepts are organized, Figure 10-4 illustrates where collaborative filtering and its subtypes fall into the broader classification of recommender systems.

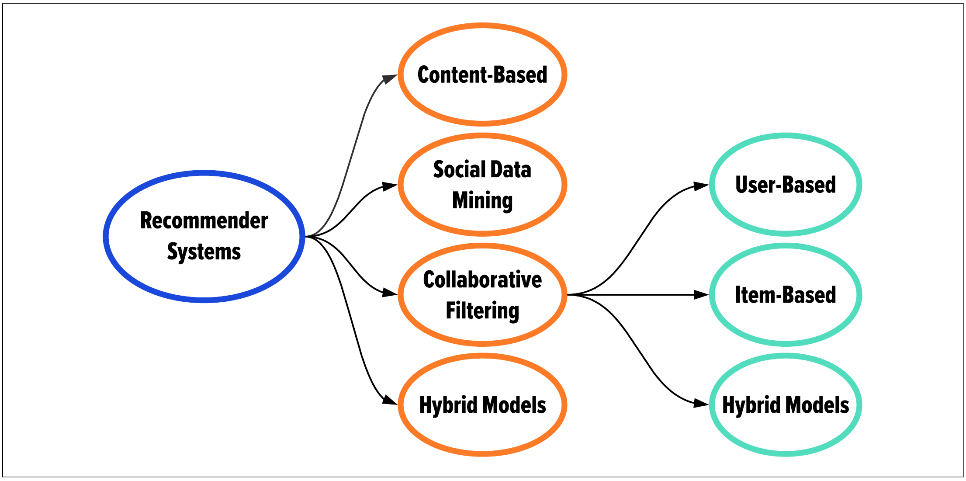



*Figure 10-4. A summary of problem classifications in the general space of recommender systems*

*Content-based* recommender systems are focused only on the preferences of the user. New recommendations are made to the user from similar content according to the user's previous choices.

The second class of recommenders, called *social data mining*, describes systems that do not need any input from a user. They rely solely on popular historical trends from the community to make recommendations to a new user.

Collaborative filtering is different from content-based or social data mining in that it combines individual and community preferences. The class of collaborative filtering approaches focuses on combining an individual's interest with the historical preferences of a community of similar users. Last, *hybrid models* are a group of recommender systems that mix and match techniques from the other three classes.

The most popular class of collaborative-filtering techniques, item-based collaborative filtering, predates the Netflix Prize we mentioned at the beginning of this chapter. Item-based collaborative filtering is one of the most robust techniques of recommendation systems of all time; it was originally invented and used by Amazon in 1998.[2] The first publication of the technique occurred in 2001.[3]

## Collaborative Filtering with Graph Data

The ability to tailor the recommendation of certain content according to the preferences of people like you describes two classes of recommendation systems: user-based collaborative filtering and item-based collaborative filtering.

*User-based collaborative filtering*
: User-based collaborative filtering finds similar users who share the same rating patterns as the active user to recommend new content.

*Item-based collaborative filtering*
: Item-based collaborative filtering finds similar items according to how users rated those items to recommend new content.

The data we will be introducing later in the chapter contains users who have rated movies. Figure 10-5 shows the different types of collaborative filtering in a model of users who rated movies.

Figure 10-5 shows how to use a graph of movie ratings to recommend new content to you. The left side of Figure 10-5 shows how to walk through the graph to perform user-based collaborative filtering and recommend new content to you. The right side of Figure 10-5 shows how to walk through the graph to perform item-based collaborative filtering and recommend new content to you.

The basic difference between user-based and item-based comes down to what each technique computes as similar. User-based collaborative filtering computes similar users, whereas item-based collaborative filtering computes similar items. Both techniques use their respective similarity scores to create a recommendation. The tasks for user-based collaborative filtering are to first compute similar users and then predict ratings of new content. The tasks for item-based collaborative filtering are to first compute similarity between items and then predict ratings of new content.

---

2 Gregory D. Linden, Jennifer A. Jacobi, and Eric A. Benson, Collaborative recommendations using item-to-item similarity mappings, U.S. Patent No. 6,266,649, filed July 24, 2001.

3 3 Badrul Munir Sarwar, George Karypis, Joseph Konstan, and John Riedl, "Item-Based Collaborative Filtering Recommendation Algorithms," WWW '01: Proceedings of the 10th International Conference on World Wide Web, Hong Kong Convention and Exhibition Center, May 1–5, 2001 (New York: ACM, 2001), 285–95. *https://doi.org/10.1145/371920.372071*.

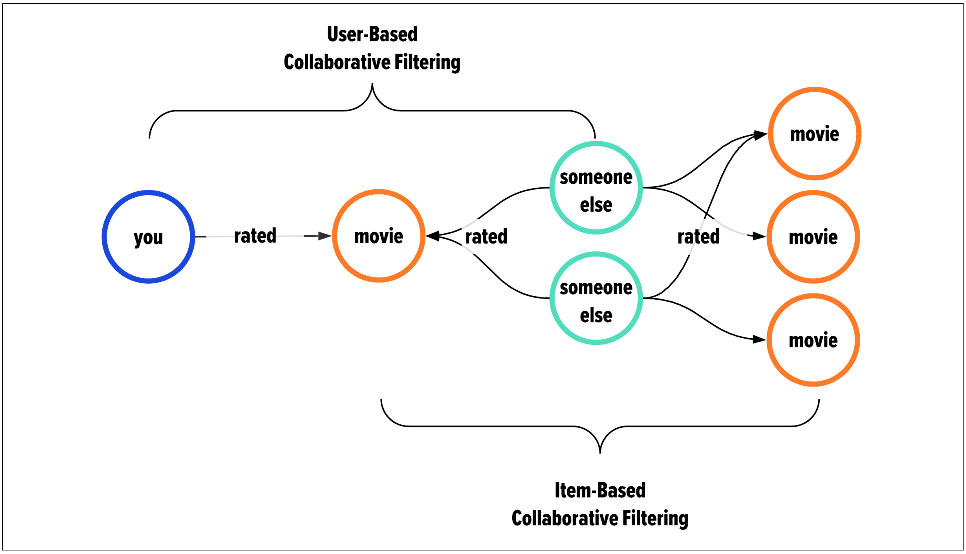



*Figure 10-5. The two types of collaborative filtering we will be using in the dataset of users and movie ratings*

We will be using item-based collaborative filtering in all of our upcoming examples in Chapters 10 and 12. The patterns you learn from exploring item-based collaborative filtering in this chapter and the production implementation process outlined in Chapter 12 show you the path forward for expanding your use of collaborative filtering to include other techniques, such as user-based.

## Recommendations via Item-Based Collaborative Filtering with Graph Data

When using a graph, the general process for using item-based collaborative filtering is as follows:

1. Input: Get a user's most recently rated, viewed, or purchased items
2. Method: Find similar items according to historical rating, viewing, or purchasing patterns
3. Recommend: Deliver different content according to a scoring model

The process above can be generalized for any system, but our upcoming examples will be about movies.

Using our movie data, which we'll introduce about four pages from now, the input will be an individual user (you) and a movie you rated. The model will use item-based collaborative filtering to find similar movies according to the rating patterns

observed in the data. The recommended content will use a scoring model to rank the recommendations. Figure 10-6 shows each of these steps.

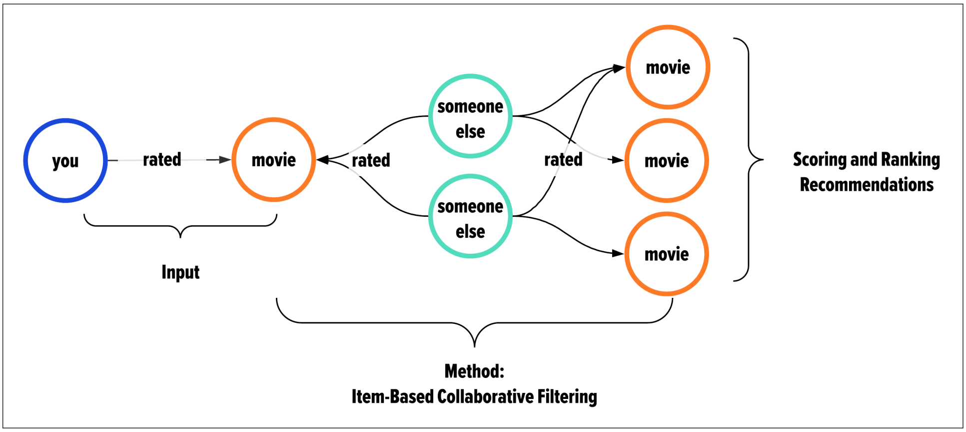



*Figure 10-6. An illustration of the input, model, and recommendation steps for our item-based collaborative-filtering graph model*

The tricks to these approaches lie in how you rank the recommended content.

## Three Different Models for Ranking Recommendations

We will be illustrating three different ways to score and rank the recommendations in our examples. They will be basic path counting, a Net Promoter Score, and a normalized Net Promoter Score.

Let's walk through each of these three approaches before we dive into the data.

### Path counting

One of the simplest ways to use a graph structure for an item-based recommendation system is to count. Specifically, you want to count the 5-star ratings from the users who also rated the input movie. Figure 10-7 shows what we mean.

Figure 10-7 shows how we would use path counting to rank the movies in the recommendation set; we bolded the two paths that reached `Movie C` to show you one of the three calculations. Let's walk through how we reached each of the scores shown on the far right.

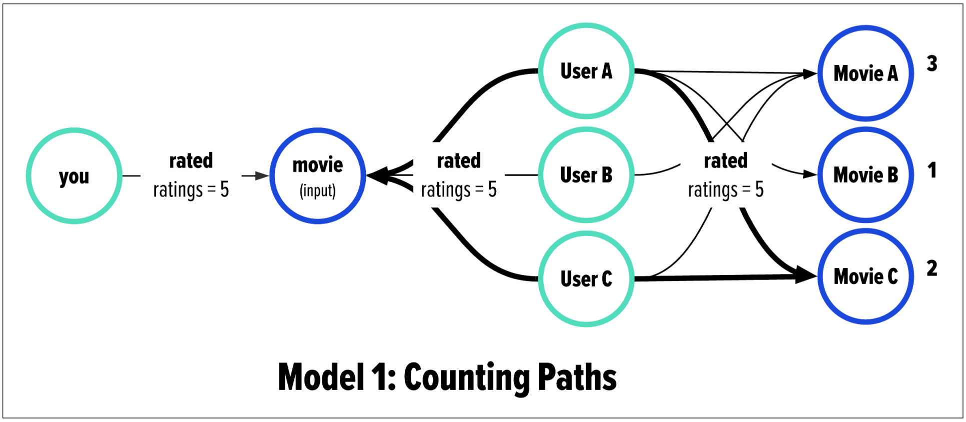



*Figure 10-7. An illustration of how to use path counting to rank the recommendation set when using item-based collaborative filtering*

The results of Figure 10-7 show `Movie A` is the top choice with a score of three. We reach a score of three because there are three 5-star ratings of `Movie A` in total, from `Users A`, B\`, and `C`. The second-ranked movie is `Movie C`, which received two 5-star ratings, one each from `Users A` and `C`. The third-ranked movie is `Movie B` with a score of one, which represents the 5-star rating from `User A`.

The final ordering of the recommendation set is: `Movie A`, `Movie C`, `Movie B`.

Counting the paths of 5-star ratings is a great place to start with item-based collaborative filtering in graphs. Hopefully, this first example is helping you to see how item-based collaborative filtering works within a graph structure.

Let's move on to a slightly more advanced scoring model.

#### Net Promoter Score–inspired metric

The Net Promoter Score (NPS) is a very popular metric that uses a scale to quantify how likely somebody is to recommend an item to a friend. For this next example, we wanted to create a metric inspired by NPS to balance the 5-star ratings from our first model with the dislikes. We are going to take the same approach with this next score. We will create a score for a movie by balancing how much it is liked with how much it is disliked.

Let's first look at how we will calculate our NPS-inspired metric from our data. Figure 10-8 shows the equation for calculating a movie's NPS.

$$NPS_m = \Sigma(ratings > 4) - \Sigma(ratings \leq 4)$$

*Figure 10-8. The equation for the Net Promoter Score (NPS) for a movie*

We will count all of the positive ratings for a movie and then subtract all of the negative ones. For our data, we consider a rating above four to indicate a liked movie and a rating below or equal to four to indicate a neutral or disliked movie.

We didn't show any edges in our first model that were not 5-star ratings. To calculate an NPS, we will need to include those edges through our traversal.

We don't want to get overly complicated because we just want to give you an idea of how the NPS works, so we are going to keep it simple by showing you two types of edges in our next example. In Figure 10-9, the thick bolded edges can be thought of as ratings greater than 4 (likes), and the dashed thin edges can be thought of as ratings below 4 (dislikes). The NPS for each movie is shown on the far right.

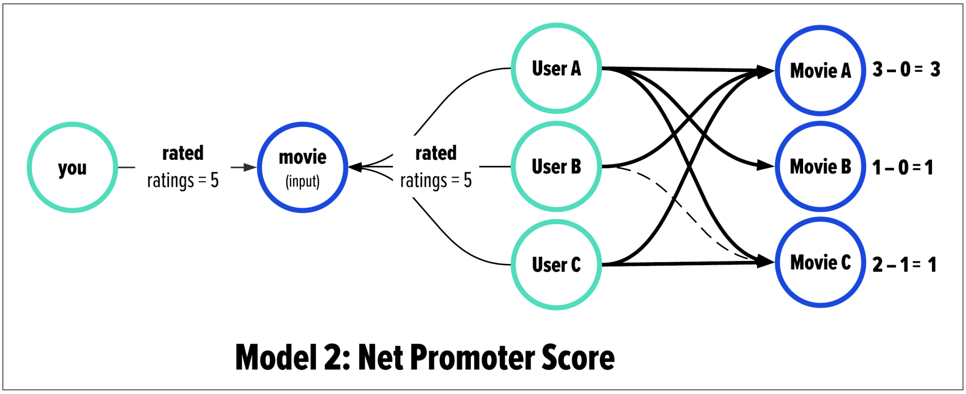



*Figure 10-9. An illustration of how to use our NPS-inspired metric to rank the recommendation set when using item-based collaborative filtering*

The final ordering of the recommendation set from Figure 10-9 is different than before: `Movie A` is the highest-rated movie, but `Movie B` and `Movie C` are tied. This example shows you how the NPS gives us a different idea of how liked (or disliked) a movie would be within different friendship communities.

The NPS and path-counting models converge on massively popular movies that are always highly ranked. This can be a problem in that the same movies will always be recommended; your user may lose interest if they see the same content every time they log in to your application. We recommend a normalized version of the NPS-inspired metric if you want to introduce some diversity into your recommendations.

Why would we want to normalize?

A normalized score will help you select "offbeat" movies for your users and add variety to your application. Ultimately, you may want to use both scores in your application to recommend two popular movies and one offbeat movie.

Let's take a look at how to introduce normalization as a way to address these issues.

### Normalized Net Promoter Scores

To account for overly popular movies, we can normalize a movie's NPS by the total number of ratings it has received. Figure 10-10 shows how we can do this using a graph property: the degree of a movie.

$$NPS_{norm} = \frac{NPS_m}{degree(movie)}$$

*Figure 10-10. The equation for the normalized Net Promoter Score (NPS norm) for a movie*

Figure 10-10 shows the third model we will be using in our examples. To arrive at the final score of a movie, we will take its NPS and then divide it by the total number of ratings it has received. For example, a really popular movie may get 50 likes out of 100 ratings, which would give it a score of 0.5. An offbeat movie may get 20 likes out of 25 ratings, giving it a score of 0.8. We want to give our input users a chance to see offbeat recommendations.

Figure 10-11 shows how we will be using the normalized NPS in our upcoming examples.

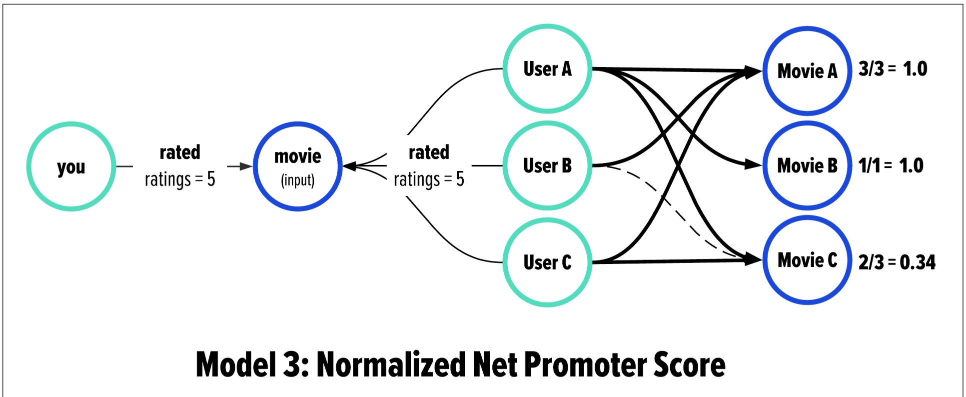



*Figure 10-11. An illustration of how to use a normalized NPS-inspired score to rank the recommendation set when using item-based collaborative filtering*

Figure 10-11 divides each movie's NPS according to its total number of ratings.

Let's step through how we calculated each movie's score in Figure 10-11. Movie A had an NPS of 3 and was rated three times; the final score for Movie A is `3/3 = 1.0`. Movie B had an NPS of 1 and was rated once; the final score for Movie B is `1/1 = 1.0`. Movie C had an NPS of 1 and was rated three times; the final score for Movie C is `1/3 = 0.3334`.

Generally, we are showing how an offbeat movie can be just as highly recommended as a very popular movie, allowing for some diversity in our movie recommendations.

Now that you have an idea of where we are going, let's present the data model we will be using for this example.

# Movie Data: Schema, Loading, and Query Review

There are two very popular open source datasets about movies that we are going to use: MovieLens[4] and Kaggle.[5] We selected the MovieLens dataset so that we could use a very diverse and well-documented dataset of user ratings of movies. The Kaggle dataset augments the MovieLens data with details and actors for each movie.

We have provided all the details as to how we matched, merged, and modeled these data sources in Chapter 11. For this chapter, we want to jump to using the data in development mode so that we can build our recommendation queries.

## Data Model for Movie Recommendations

The data integration process between the MovieLens and Kaggle sources that we outline in Chapter 11 created the development schema we will be using in our examples. Using the Graph Schema Language (GSL), the development schema is shown in Figure 10-12.

> The data model in Figure 10-12 has a lot of detail. If you prefer to understand how we arrived at our data model before we use it, we recommend jumping over to Chapter 11, where we outline in depth how we merged the two data sources and created this model. The process was too long and involved to go through now. And the topic of entity resolution deserves its own separate discussion.

4 F. Maxwell Harper and Joseph A. Konstan, "The MovieLens Datasets: History and Context," *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, no. 4 (2016): 19, *https://doi.org/10.1145/2827872*.

5 Stephane Rappeneau, "350 000+ Movies from themoviedb.org," Kaggle, July 19, 2016, *https://www.kaggle.com/stephanerappeneau/350-000-movies-from-themoviedborg*.

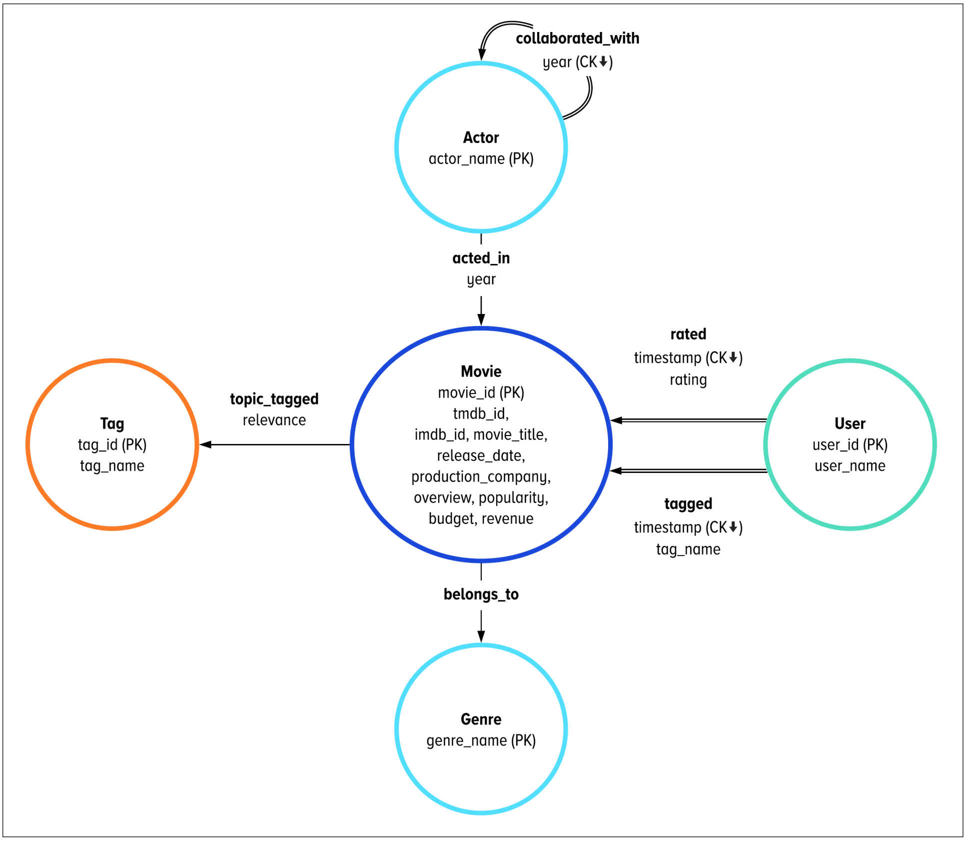



*Figure 10-12. The development schema for our merged database about movies*

Figure 10-12 shows that our data has five vertex labels: `Movie`, `User`, `Genre`, `Actor`, and `Tag`. The partition key for each vertex label is denoted with a (PK) next to the property name. There are three edge labels that will each have exactly one edge between the vertices it connects: `acted_in`, `belongs_to`, and `topic_tagged`. The GSL notation of a single line indicates that an actor acts in a specific movie only once, a movie belongs to a specific genre only once, and a movie is tagged for a specific topic only once. There are three edge labels that have many edges between the vertices they connect: `rated`, `tagged`, and `collaborated_with`. The GSL notation of a double line and property with a clustering key (CK) indicates that a user can rate a specific movie many times, a user can tag a specific movie many times, and an actor can collaborate with another actor many times.

We hope you are used to looking at images of graph schema and using the GSL to translate into schema statements. We designed this process to follow how ERDs provide a programmatic way to translate a conceptual model into schema code.

## Schema Code for Movie Recommendations

From Figure 10-12, we see that there are five vertex labels. The schema code for those vertex labels is shown in Example 10-1.

*Example 10-1.*

```
schema.vertexLabel("Movie").
       ifNotExists().
       partitionBy("movie_id", Bigint).
       property("tmdb_id", Text).
       property("imdb_id", Text).
       property("movie_title", Text).
       property("release_date", Text).
       property("production_company", Text).
       property("overview", Text).
       property("popularity", Double).
       property("budget", Bigint).
       property("revenue", Bigint).
       create();

schema.vertexLabel("User").
       ifNotExists().
       partitionBy("user_id", Int).
       property("user_name", Text). // Augmented, Random Data by the authors
       create();

schema.vertexLabel("Tag").
       ifNotExists().
       partitionBy("tag_id", Int).
       property("tag_name", Text).
       create();

schema.vertexLabel("Genre").
       ifNotExists().
       partitionBy("genre_name", Text).
       create();

schema.vertexLabel("Actor").
       ifNotExists().
       partitionBy("actor_name", Text).
       create();
```

Fun fact about the data types you see in Example 10-1: the total revenue for the movie *Avatar* was so high that we had to change the data type for `budget` and `revenue` from Int to Bigint. Way to go, James Cameron.

During the ETL (extract-transform-load) process in Chapter 11 that we wrote to merge the MovieLens and Kaggle data sources, we used Python's Faker library to randomly generate names for our users. This data is not in any way associated to the users of the MovieLens project; the names are completely random.

From Figure 10-12, we see that there are six edge labels. The schema code for those edge labels is shown in Example 10-2.

*Example 10-2.*

```
schema.edgeLabel("topic_tagged").
       ifNotExists().
       from("Movie").
       to("Tag").
       property("relevance", Double).
       create()

schema.edgeLabel("belongs_to").
       ifNotExists().
       from("Movie").
       to("Genre").
       create()

schema.edgeLabel("rated").
       ifNotExists().
       from("User").
       to("Movie").
       clusterBy("timestamp", Text). // Makes the ISO 8601 standard easier to use
       property("rating", Double).
       create()

schema.edgeLabel("tagged").
       ifNotExists().
       from("User").
       to("Movie").
       clusterBy("timestamp", Text). // Makes the ISO 8601 standard easier to use
       property("tag_name", Text).
       create()

schema.edgeLabel("acted_in").
       ifNotExists().
       from("Actor").
       to("Movie").
       property("year", Int).
       create()

schema.edgeLabel("collaborated_with").
       ifNotExists().
       from("Actor").
```

```
        to("Actor").
        clusterBy("year", Int).
        create()
```

We did the data ETL of matching and merging the MovieLens and Kaggle datasets for you. Along the way, we also formatted the new dataset so that it was ready to be loaded into DataStax Graph. One change we have made a few times throughout this book is to format time in the ISO 8601 standard to make it easier to reason about examples in a book.

Next, let's look at some of the data and at how to load the datafiles into DataStax Graph.

## Loading the Movie Data

We are continuing to use the bulk loading functionality that comes with DataStax Graph so that the datasets can be loaded into the underlying tables in Cassandra as quickly as possible.

Part of that process requires formatting the datafiles to match the schema in DataStax Graph. We already did that work for you. The work involved writing files that matched the property names for the vertex and edge schema that we just created in the last section.

Let's walk through loading the vertex data and then show the same for our edges.

### Loading the vertices

The first thing we want to show you is how we formatted some of the vertex data. We are going to look at three of the five files. Figure 10-13 shows the first three lines (including the header) of the movie data that we merged and created for this example. We trimmed the `overview` description from this book. The file and loaded data does contain the full overview of the movie.

| movie_id | tmdb_id | imdb_id | movie_title | release_date | production_company | popularity | budget | revenue | overview |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 862 | 0114709 | Toy Story (1995) | 1995-10-30T00:00:00 | Pixar Animation | 8.644397 | 30000000 | 373554033 | Led by Woody Andy ... |
| 2 | 8844 | 0113497 | Jumanji (1995) | 1995-12-15T00:00:00 | TriStar Pictures | 3.594827 | 65000000 | 262797249 | When siblings Judy and Peter discover ... |

*Figure 10-13. The header line and first two movies in our data*

The header of a vertex file must match the property names found in the DataStax Graph schema. The first line of Figure 10-13 confirms this to be the case, as we see that the header values match the property key names we defined in Example 10-1.

Additionally, we wanted to make it easier to query and reason about time in this data, so we transformed the timestamps from epoch to the ISO 8601 standard and stored them as a string. You can see this in the fifth column from the left in Figure 10-13. This is not recommended for production as it introduces an extra storage cost, but it makes it easier to reason about the data when playing around with it.

Let's look at two more datafiles that we loaded for this example; Table 10-1 shows some of the actors in the dataset.

*Table 10-1. The first five actors from the `Actor.csv` file*

| actor_name | gender_label |
|---|---|
| Turo Pajala | unknown |
| Susanna Haavisto | unknown |
| Matti Pellonpää | male |
| Eetu Hilkamo | unknown |
| Kati Outinen | female |

Last, Table 10-2 shows some of the users we loaded into the database. We augmented the users with fake names; they are not in any way related to the MovieLens users.

*Table 10-2. The first four users from the `User.csv` file*

| user_id | user_name |
|---|---|
| 1 | Laura Pace |
| 2 | James Thornton |
| 3 | Timothy Fernandez |
| 4 | Stacy Roth |

We created one `csv` file per vertex label for these examples, for a total of five files. We did this extra work so that it would be very easy to load the data directly into DataStax Graph. Example 10-3 shows the five commands needed to load the data.

*Example 10-3.*

```
dsbulk load -g movies_dev -v Movie
            -url "Movie.csv" -header true
dsbulk load -g movies_dev -v User
            -url "User.csv" -header true
dsbulk load -g movies_dev -v Tag
            -url "Tag.csv" -header true
dsbulk load -g movies_dev -v Genre
            -url "Genre.csv" -header true
dsbulk load -g movies_dev -v Actor
            -url "Actor.csv" -header true
```

For each of the five vertex labels, Table 10-3 shows the total number of vertices from each file that are processed by the bulk loading tool.

*Table 10-3. The total number of vertices from each file that will be inserted in our example's graph*

| | |
|---|---|
| 260860 | actor_vertices.csv |
| 1170 | genre_vertices.csv |
| 329470 | movie_vertices.csv |
| 1129 | tag_vertices.csv |
| 138494 | user_vertices.csv |

Now that we have loaded all of our vertices into our development graph, we can connect them together with the edge datasets.

#### Loading the edges

The last concept we want to show you is how we formatted some of the edge data. We are going to look at three of the six files. Table 10-4 shows the first three lines (header included) of the rating data that we created for this example.

*Table 10-4. The first two ratings by users from the `rated_100k_sample.csv` file*

| User_user_id | Movie_movie_id | rating | timestamp |
|---|---|---|---|
| 1 | 2 | 3.5 | 2005-04-02 18:53:47 |
| 1 | 29 | 3.5 | 2005-04-02 18:31:16 |

As you have already seen a few times in the book, the header line is the most important concept for formatting your files to match an edge label's schema. Table 10-4 shows how the data about user ratings matches to the schema in DataStax Graph. The header line has to have the property names used by the edge table in DataStax Graph; that is why the first column is labeled `User_user_id` and the second column is named `Movie_movie_id`. You can obtain this information in three different ways: (1) through the Studio schema inspection tool, (2) via `cqlsh`, or (3) by following the naming conventions of the edge tables into Cassandra tables.

Next, Table 10-5 shows the first three lines (header included) of the actor edges that we created for this example.

*Table 10-5. The first two connections from actors to movies from the `acted_in.csv` file*

| Actor_actor_name | year | Movie_movie_id |
|---|---|---|
| Turo Pajala | 1988 | 4470 |
| Susanna Haavisto | 1988 | 4470 |

Table 10-5 shows two edges from actors to movies in the database. We see that the actors `Turo Pajala` and `Susanna Haavisto` acted in the movie with a `movie_id` of `4470` in 1988.

Last, let's look at the collaborator edges we created for this example, in Table 10-6.

*Table 10-6. The first two actor collaborations from the `collaborator.csv` file*

| in_actor_name | year | out_actor_name |
| --- | --- | --- |
| Turo Pajala | 1988 | Susanna Haavisto |
| Turo Pajala | 1988 | Matti Pellonpää |

Table 10-6 shows two edges about actors who appeared in the same movie. We see that `Turo Pajala` and `Susanna Haavisto` acted in the movie in 1988 and therefore are listed as collaborators. We expected this based on what we saw in our actor data.

If you would like, you can spend time now looking at all of the edge files that accompany this text. Since we have done this a few times now, we are going to move forward to loading the edges into the database.

To load all of the edges, we can use the bulk loading command-line tool to load them into tables in Apache Cassandra. Example 10-4 shows the six commands needed to load this data.

*Example 10-4.*

```
dsbulk load -g movies_dev -e belongs_to -from Movie -to Genre
            -url belongs_to.csv -header true
dsbulk load -g movies_dev -e topic_tagged -from Movie -to Tag
            -url topic_tag_100k_sample.csv -header true
dsbulk load -g movies_dev -e rated -from User -to Movie
            -url rated_100k_sample.csv -header true
dsbulk load -g movies_dev -e tagged -from User -to Movie
            -url tagged.csv -header true
dsbulk load -g movies_dev -e acted_in -from Actor -to Movie
            -url acted_in.csv -header true
dsbulk load -g movies_dev -e collaborated_with -from Actor -to Actor
            -url collaborator.csv -header true
```

For each of the six edge labels, Table 10-7 shows the total number of lines in each file that are processed by the bulk loading tool.

*Table 10-7. The total number of edges from each file to be loaded into our example graph*

| | |
| --- | --- |
| 836408 | acted.csv |
| 2706175 | collaborator.csv |
| 523689 | contains_genre.csv |

| | |
|---|---|
| 11709769 | movie_topic_tag.csv |
| 100000 | rated.csv |
| 465321 | tagged.csv |

From here, we are ready to query this data in DataStax Graph. We want to start with some basic exploration of the data. For review, we are going to do three exploration queries that repeat the first three query patterns we taught in this book: walking through neighborhoods, trees, and paths in the movie data.

There are so many more interesting queries we could do with this data. We hope you explore what is possible in development mode in your notebook by applying the techniques from Chapter 4, Chapter 6, and Chapter 8 to answer other interesting questions.

## Neighborhood Queries in the Movie Data

After loading a new dataset into your graph, the first queries you will want to try explore first neighborhoods around a single vertex. Let's recall the basics of walking around the first neighborhood of your data to show a specific user's movie ratings.

The first query we are going to explore in this data is: for user `134558`, show me all movies rated by this user, with each movie's rating. Example 10-5 shows this query in Gremlin.

*Example 10-5.*

```
dev.V().has("User","user_id", 134558).   // WHERE: start at the user
        outE("rated").                    // JOIN: walk out to all rated edges
        project("movie", "rating", "timestamp"). // CREATE a json payload
          by(inV().values("movie_title")).       // JOIN and SELECT the movie title
          by(values("rating")).                  // SELECT the edge's rating
          by(values("timestamp"))                // SELECT the edge's timestamp
```

The first three results of Example 10-5 are shown in Example 10-6.

*Example 10-6.*

```
{
  "movie": "Toy Story (1995)",
  "rating": "3.5",
  "timestamp": "2013-06-08 08:22:47"
},
{
  "movie": "GoldenEye (1995)",
  "rating": "3.5",
  "timestamp": "2013-06-08 08:25:13"
},
```

```
{
  "movie": "Twelve Monkeys (aka 12 Monkeys) (1995)",
  "rating": "2.0",
  "timestamp": "2013-06-08 08:23:45"
},...
```

The result in Example 10-6 is a list of maps. Each map has the three keys, `movie`, `rating`, and `timestamp`, which set up in our query from Example 10-5. We selected the values for each respective key, where you also see that the ISO 8601 standard is used for representing timestamps.

Figure 10-14 shows another way to think of the first three results from Example 10-6.

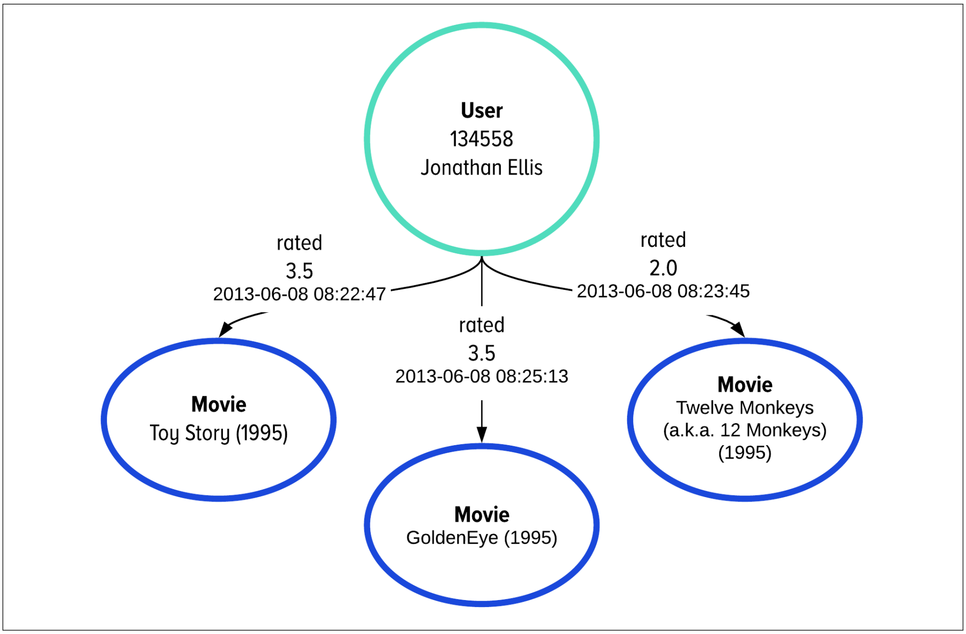



*Figure 10-14. Visualizing the first three results of Example 10-5*

More often than not, there is a little bit of manipulation you would like to do to your query results. We have practiced shaping query results many times throughout this book. Let's take one more look at how we could query the first neighborhood around user `134558` to list our user's movies by the ones they liked, disliked, or are neutral about.

### Grouping a user's movie ratings by liked, disliked, or neutral

In this next example, we want to query the first neighborhood of user `134558`. But this time we want to group the movies rated by `134558` according to whether the user liked, disliked, or was neutral about them. The rating scale in our data ranges from

0.5 to 5.0. We will say that movies with a rating of 4.5 or higher are liked. Movies with a rating between 3.0 and 4.5, but not including 4.5, will be considered neutral. Movies with a rating between 0 and 3.0, but not including 3.0, will be considered disliked. Yes, this is a different rating system than the models we walked through before; we are using this example to teach concepts in shaping neighborhood results. Eventually, we will get back to recommendations.

Let's look at how to do this in Gremlin in Example 10-7.

> If you prefer, the step `choose()` can replace `coalesce()` in Example 10-7.

*Example 10-7.*

```
1 dev.V().has("User","user_id", 134558). // WHERE: start at the user
2         outE("rated").                  // JOIN: walk to the "rated" edge
3         group().                        // CREATE: make a group
4           by(values("rating").          // SELECT KEYS: according to the ratings
5              coalesce(__.is(gte(4.5)).constant("liked"),   // KEY 1: "liked"
6                       __.is(gte(3.0)).constant("neutral"), // KEY 2: "neutral"
7                        constant("disliked"))).             // KEY 3: "disliked"
8           by(inV().values("movie_title").fold()) // SELECT VALUES: the values
```

Let's walk through each step of Example 10-7 before we look at the results in Example 10-8. Lines 1 and 2 in Example 10-7 start at user `134558` and walk out to each of the user's ratings. Line 3 in Example 10-7 creates a group. A group in Gremlin always has two components: keys and values. The first `by()` step wraps lines 4 through 7 and sets up the keys. The second `by()` step is on line 8 and determines the values for the group. The keys will be "liked," "neutral," or "dislike." We use the `coalesce` step in Gremlin like an `if/elif/else` statement to determine into which key the user's rating will be grouped. Line 5 filters all ratings with a value of 4.5 or higher into the "liked" group. After that filter, the remaining edges flow to the next filter on line 6, which grabs all ratings of 3.0 or higher for the "neutral" group. All other edges will have a rating under 3.0 and will go into the "disliked" key.

Line 8 of Example 10-7 is the last step for shaping the query results. For each object in this map, we want the value to be the movie title. Therefore, we have to walk from the edge into the movie vertex and grab the movie title.

Example 10-8 displays the first three movies for each key.

*Example 10-8.*

```
{
    "neutral": [
                        "GoldenEye (1995)",
                        "Babe (1995)",
                        "Apollo 13 (1995)",
                        ...
                        ],
    "liked": [
                        "Braveheart (1995)",
                        "Shawshank Redemption The (1994)",
                        "Forrest Gump (1994)",
                        ...
                        ],
    "disliked": [
                        "Twelve Monkeys (aka 12 Monkeys) (1995)",
                        "Stargate (1994)",
                        "Ace Ventura: Pet Detective (1994)",
                        ...
                        ]
}
```

The first two example queries in this chapter give you an idea of how to walk through the neighborhoods of data in the movie database. We hope they were a good review of the different queries we have been teaching throughout this book.

The next main example is to walk through a tree within this dataset.

## Tree Queries in the Movie Data

As we talked about when we did the tree queries through our sensor data, your data's branching factor can get out of hand really quickly. That remains true for the data that we loaded for this example.

We went through the difficulty of integrating the Kaggle dataset into our database so that we could have some type of tree to query in our data. We like to think of the following query as looking at an actor's "family tree" of collaborators.

The tree we want to find in our data starts with Kevin Bacon and finds a lineage of actors he worked with. We kept it simple and limited the query in two ways. First, we wanted to consider only his collaborations from 2009 onward. And because everyone is somehow connected to Kevin Bacon, we wanted to look only three levels deep into the tree.

Let's look at the query in Example 10-9.

*Example 10-9.*

```
1 dev.V().has("Actor", "actor_name", "Kevin Bacon").as("Mr. Bacon").
2        repeat(outE("collaborated_with").has("year", gte(2009)).as("year").
3               inV().as("collaborated_with").
4               simplePath()).
5        times(3).
6        path().
7          by("actor_name").
8          by("year")
```

The query in Example 10-9 starts at Kevin Bacon and walks out to all of his collaborators starting in 2009. This repeats three times, where we eliminate repeating paths through the data with `simplePath()` on line 4. After walking three layers deep, we shape the results on lines 6 through 8 by returning the actor's name from the vertices and the year from the edges in the path object.

Example 10-10 shows the first two results from Example 10-9.

*Example 10-10.*

```
{
    "labels": [["Mr. Bacon"],["year"],["collaborated_with"],
               ["year"],["collaborated_with"],
               ["year"],["collaborated_with"]],
    "objects": ["Kevin Bacon","2009","David Koechner",
                "2009","Bob Gunton",
                "2009","Gretchen Mol"]
},
    "labels": [["Mr. Bacon"],["year"],["collaborated_with"],
               ["year"],["collaborated_with"],
               ["year"],["collaborated_with"]],
    "objects": ["Kevin Bacon","2009","Renée Zellweger",
                "2010","Forest Whitaker",
                "2009","Jessica Biel"]
},...
```

Figure 10-15 visualizes the tree of results from Example 10-10.

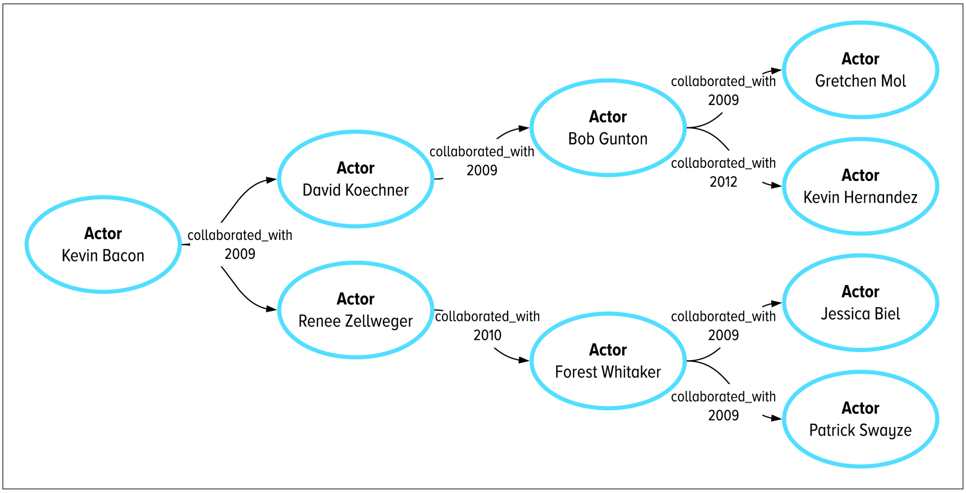



*Figure 10-15. Visualizing the actor tree of the first five results from Example 10-10*

The last query we want to explore in this data recalls the pathfinding queries we built over the Bitcoin data. Let's look at how to find paths in this dataset.

## Path Queries in the Movie Data

Every actor is connected to Kevin Bacon. Let's use this colloquialism to find paths between two actors in our dataset.

Example 10-11 uses the `collaborated_with` edges to find the first three shortest paths between Kevin Bacon and Morgan Freeman.

*Example 10-11.*

```
1 dev.V().has("Actor", "actor_name", "Kevin Bacon").as("Mr. Bacon").
2         repeat(outE("collaborated_with").as("year").
3                inV().as("collaborated_with")).
4         until(has("Actor", "actor_name", "Morgan Freeman").as("Mr. Freeman")).
5         limit(3).
6         path().
7           by("actor_name").
8           by("year")
```

Recall that the pattern of `repeat().until()` uses breadth-first search without a barrier. Therefore, when we have `limit(3)` on line 5 in Example 10-11, we are really finding the three shortest paths in this dataset that satisfy the stopping condition. As we have walked through many times throughout this book, lines 6, 7, and 8 shape the results of the path object.

Example 10-12 shows the JSON payload for Example 10-11.

*Example 10-12.*

```
{
    "labels": [["Mr. Bacon"],
               ["year"],["collaborated_with"],
               ["year"],["collaborated_with"]],
    "objects": ["Kevin Bacon",
                "1979","Julie Harris",
                "1990","Morgan Freeman"]
},{
    "labels": [["Mr. Bacon"],
               ["year"],["collaborated_with"],
               ["year"],["collaborated_with"]],
    "objects": ["Kevin Bacon",
                "1982","Mickey Rourke",
                "1989","Morgan Freeman"]
},{
    "labels": [["Mr. Bacon"],
               ["year"],["collaborated_with"],
               ["year"],["collaborated_with"]],
    "objects": ["Kevin Bacon",
                "1983","Ellen Barkin",
                "1984","Morgan Freeman"]
}
```

For fun, we also wanted to take a look at the results in their graph structure. Figure 10-16 shows the three paths from Example 10-12.

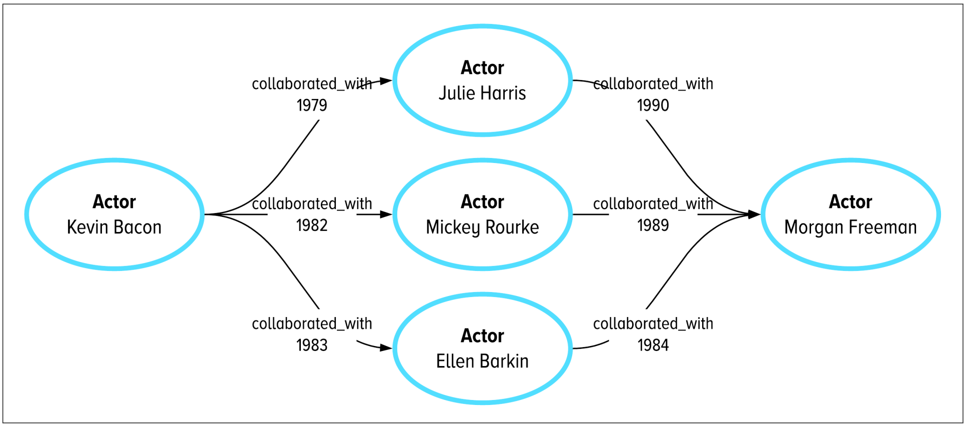



*Figure 10-16. Visualizing the first three shortest collaborator paths from the results of Example 10-11*

We hope you found the four queries in this section to be a helpful review of the query concepts we have been teaching throughout this book. We leave it up to you to turn these into production queries; we won't be doing that in the production recommendation chapter.

Now let's return to the topic of this chapter: recommendation systems. The next section will build up different Gremlin queries to show you how to do collaborative filtering.

# Item-Based Collaborative Filtering in Gremlin

We have built up our use case, defined collaborative filtering, seen some examples, and explored our data. The last part of this chapter focuses on performing item-based collaborative filtering to recommend new movies to a user in our data. We are going to show you three different ways to do that in a development environment.

Let's get started with the query and results for the first approach to item-based collaborative filtering with graph data.

## Model 1: Counting Paths in the Recommendation Set

The first way that we will be recommending movies to a user will follow the basic path-counting approach. The general process of walking through your graph data to do this is outlined in Example 10-13.

*Example 10-13.*

```
For a specific user
    Walk to the last movie they rated
    Walk to all users who highly rated that movie
    Walk to all movies highly rated by these users
    Group and count all movies in the recommendation set
    Sort the movies by frequency, in descending order
    The top movies form the recommendation set
```

The pseudocode in Example 10-13 outlines how we are going to walk through our graph data for our first collaborative-filtering example. This first approach essentially counts how often a movie shows up in the recommendation set. The movies with the highest scores would be considered the most likely recommendations based on the user's most recent rating.

Example 10-14 shows the Gremlin query from the approach outlined in Example 10-13.

*Example 10-14.*

```
dev.V().has("User","user_id", 694).   // look up a user
   outE("rated").                     // traverse to all rated movies
     order().by("timestamp", desc).   // order all edges by time
     limit(1).inV().                  // traverse to the most recent rated movie
     aggregate("originalMovie").      // put this movie in a collection
   inE("rated").has("rating", gt(4.5)).outV(). // users who rated this movie 5
```

```
7    outE("rated").has("rating", gt(4.5)).inV(). // the full recommendation set
8    where(without("originalMovie")).   // remove the original movie
9    group().                           // create a map of the recommendations
10     by("movie_title").               // an entry's key is the movie title,
11     by(count()).                     // the value will be the total # of ratings
12   unfold().                          // unfold all map entries into the pipeline
13   order().                           // order the results
14     by(values, desc)                 // by their count, descending
```

Let's step through Example 10-14. Lines 1 and 2 look up a specific user vertex in the graph and traverse out to all of the user's ratings. Lines 3 and 4 sort the ratings by time and traverse through only the most recent rating to the movie vertex. We store this movie in a collection on line 5 so that we can later remove the movie from the recommendation options. On line 6, we walk from the movie to all users who have rated that movie with a 5. We traverse from these users to all movies that they have also rated with a 5. At this point, we remove the original rated movie on line 8.

We start formatting our result set on line 9, where we create a map. Line 10 shows that the keys of the map will be `movie_title`. Line 11 shows that the values will be the total number of traversers that have reached that movie. Because this is one map, we unfold all entries in the map into the traversal pipeline on line 12. Lines 13 and 14 order the individual maps according to their values.

Figure 10-17 shows the top five recommended movies from Example 10-14.

| index ↑ | keys | values |
|---|---|---|
| 0 | Shawshank Redemption The (1994) | 24 |
| 1 | Forrest Gump (1994) | 22 |
| 2 | Apollo 13 (1995) | 21 |
| 3 | Jurassic Park (1993) | 21 |
| 4 | Schindler's List (1993) | 20 |

*Figure 10-17. The top five results from Example 10-14*

In Figure 10-17, we see that *The Shawshank Redemption* has a score of 24, *Forrest Gump* has a score of 22, and *Apollo 13* has a score of 21.

This first model is based only on tracing 5-star ratings through our sample data. Let's see how to make the ranking algorithm a bit more sophisticated.

## Model 2: NPS-Inspired

The second way we want to recommend movies uses a version of the Net Promoter Score (NPS). We will consider a rating of 4 or higher to represent a liked movie and a

rating less than 4 to represent a disliked movie. We will add this into the same process that we outlined before when we process the user's ratings. Let's look at the pseudocode in Example 10-15 to understand how we will be walking through the graph's data.

*Example 10-15.*

```
For a specific user
    Walk to the last movie they rated
    Walk to all users who highly rated that movie
    Walk to all outgoing rating edges
        For each edge
            If the rating is 4 or higher,
                Store 1 in the traverser's sack
            If the rating is less than 4,
                Store -1 in the traverser's sack
    Walk into all movies
    Group all movies in the recommendation set
    For each movie in the group,
        Calculate the movie's NPS by adding all the traversers' sacks
    Sort the movies by NPS, in descending order
    The top movies form the recommendation set
```

The approach outlined in Example 10-15 is very similar to our first model. The only addition occurs when we traverse out from the user set to all of the user's ratings because we are including all ratings. If the rating is 4 or greater, we will add one to the overall NPS. If the rating is less than 4, we will subtract one from the overall NPS.

We will use the `sack()` step in Gremlin to do this as efficiently as possible. We will allow each traverser to walk through the data and keep track of the edge's rating along the way in its sack. Then we will group all of the traversers together as we did before. But instead of counting the total number of traversers that reached a movie, we will add together the values stored in their sack to create an NPS. After that, we will follow the same ordering process that we saw in the last query.

Example 10-16 shows the Gremlin query from the approach outlined in Example 10-15.

*Example 10-16.*

```
dev.withSack(0.0).                           // use sack to calculate NPS
   V().has("User","user_id", 694).
   outE("rated").
     order().by("timestamp", desc).
     limit(1).inV().
     aggregate("originalMovie").
   inE("rated").has("rating", gt(4.5)).outV().
   outE("rated").
```

```
9      choose(values("rating").is(gte(4.0)), // testing the rating value
10            sack(sum).by(constant(1.0)),   // add 1 if user liked the movie
11            sack(minus).by(constant(1.0))).// subtract 1 if disliked
12     inV().
13   where(without("originalMovie")).
14   group().
15     by("movie_title").
16     by(sack().sum()).                     // NPS: sum all sack values
17   unfold().
18   order().
19     by(values, desc)
```

Let's step through Example 10-16. The first new piece is the use of `withSack(0.0)` on line 1, like we saw in the last chapter on calculating weighted paths. Lines 2 through 8 follow the same setup as the first query we walked through in this section.

Line 9 of Example 10-16 shows how we start to set up for calculating the NPS based on a user's rating. We are showing you how to use `choose(condition, true, false)` semantics with Gremlin. The condition is on line 9 and checks if the edge's rating is greater than or equal to 4. If this is true, line 10 shows how we add `1.0` into the traverser's sack. If the condition on line 9 is false, we subtract `1.0` from the traverser's sack. On line 12, the traversers move to all of the movies for the ratings, and line 13 removes the original movie.

Lines 14 through 19 follow the same grouping and sorting process as before, but with one change. Line 16 shows that the value for a movie in the map is the sum of all of the traverser's sacks that arrived at that movie. We will be adding together a bunch of 1s and/or –1s. The top five recommended movies from Example 10-16 are shown in Figure 10-18.

| index ↑ | keys | values |
|---|---|---|
| 0 | Fugitive The (1993) | 30.0 |
| 1 | Star Wars: Episode IV - A New Hope (1977) | 28.0 |
| 2 | Forrest Gump (1994) | 28.0 |
| 3 | Apollo 13 (1995) | 26.0 |
| 4 | Terminator 2: Judgment Day (1991) | 25.0 |

*Figure 10-18. The top five results from Example 10-16*

In Figure 10-18, we see a different set of recommendations from what we saw in Figure 10-17: *The Fugitive* has a score of 30, *Star Wars: Episode IV—A New Hope* has a score of 28, and *Forrest Gump* also has a score of 28.

This second model can still produce a repetitive set of results in which popular movies continue to show up as the main recommendations. Let's take this one step further and see how we can normalize the result set so we can try to find a diverse set of recommendations.

## Model 3: Normalized NPS

The final way we will use item-based collaborative filtering on our data illustrates one way to use normalization in the scoring model. We will still use a movie's NPS as we did in the last section but will ultimately divide the NPS by the total number of ratings we have observed for that movie. Example 10-17 walks through the pseudocode for how we will do this as we walk through our graph data.

*Example 10-17.*

```
For a specific user
    Walk to the last movie they rated
    Walk to all users who highly rated that movie
    Walk to all outgoing rating edges
        For each edge
            If the rating is 4 or higher, store 1 in the traverser's sack
            If the rating is less than 4, store -1 in the traverser's sack
    Walk into all movies
    Group all movies in the recommendation set
    For each movie in the group,
        Calculate its NPS
        Count all of its incoming ratings
        Divide NPS by incoming ratings
```

Example 10-17 follows the same process as when we had to calculate our NPS. However, when we create our map of recommendations, we will divide the NPS by the movie's total number of incoming ratings. Let's see how we will do this in Gremlin in Example 10-18.

*Example 10-18.*

```
dev.withSack(0.0).
   V().has("User","user_id", 694).
   outE("rated").
     order().by("timestamp", desc).
     limit(1).inV().
     aggregate("originalMovie").
   inE("rated").has("rating", gt(4.5)).outV().
   outE("rated").
      choose(values("rating").is(gte(4.0)),
             sack(sum).by(constant(1.0)),
             sack(minus).by(constant(1.0))).
      inV().
```

```
13    where(without("originalMovie")).
14    group().
15      by("movie_title").
16      by(project("numerator", "denominator"). // NPS/degree(movie)
17           by(sack().sum()).                  // this is NPS
18           by(inE("rated").count()).          // this is the degree of the movie
19         math("numerator/denominator"))       // this is how we divide them
```

Lines 1 through 15 in Example 10-18 are the same as the first 15 lines of Example 10-16, in which we calculated the NPS. The new code spans lines 16 through 19 and shows how to divide the NPS by the in-degree of the movie.

On line 16 of Example 10-18, we are filling in the values for the movies that will populate our map of results. The value that will go into the map will be the NPS for the movie divided by its total number of ratings. We create a map with only two elements by using the `project()` step. Then the math step on line 19 will divide these two values and put the result into the group. Line 17 forms the first element of the map and is the movie's NPS. Line 18 forms the second element and is the total number of incoming ratings. The first five results are shown in Figure 10-19.

| index ↑ | keys | values |
|---|---|---|
| 0 | Apocalypse Now (1979) | 0.00819672131147541 |
| 1 | Spider-Man (2002) | 0.009433962264150943 |
| 2 | Repo Man (1984) | 0.027777777777777776 |
| 3 | Juno (2007) | 0.023255813953488372 |
| 4 | Men in Black (a.k.a. MIB) (1997) | 0.005319148936170213 |

*Figure 10-19. The top five results of Example 10-18*

The results of Figure 10-19 show the first five examples of the normalized NPS. All five examples have a positive score and would be considered "liked" according to this model. There are movies with negative scores that you can explore in the Studio Notebook for this chapter.

Some of you may be wondering why we aren't showing the sorted version of Figure 10-19 and the top five recommendations. We are showing the first five instead of the top five because this final query is pushing the limits of what we can reasonably compute within a traversal, even on this small sample set. It is that additional `inE("rated").count()`, which is another full partition scan per vertex, that is making this query extremely expensive.

## Choosing Your Own Adventure: Movies and Graph Problems Edition

In order to be able to deliver item-based collaborative filtering in a production environment with real user expectations, we have gone significantly past what would be reasonable to do in real time.

So at this point, you get to choose where to go next. You have two options.

The first option is to go back and understand the data we merged for this example. Chapter 11 takes a brief side tour into how we matched the MovieLens and Kaggle data together for the model and queries you saw in this chapter. We would bet that any graph user has to walk through some form of data cleaning and merging, no matter how simple it may be. If you are interested in simple entity resolution, continue on to the next chapter.

If you want to skip the nuances of basic entity resolution, we won't blame you. In that case, jump ahead to Chapter 12 to continue with the production version of item-based collaborative filtering. In Chapter 12, we will explain why the traversals we have worked through here in development mode cannot be run in a production environment. We will walk through the last production tip for this book and show you how to deliver recommendations from item-based collaborative filtering with graph data.

CHAPTER 11

# Simple Entity Resolution in Graphs

Thinking back to our first example in this book, how do you know who your customer is in your C360 model?

Do you have a strong identifier in your dataset, like a social security number or member ID? How much do you trust those identifiers, and their source, to represent unique people with 100% accuracy?

Different industries have different tolerance levels for inaccuracy.

In healthcare, false positives can lead to misdiagnoses and potentially deadly distributions of medicine. On the other hand, if you are working with data about movies, incorrect movie resolution will lead to a less-than-seamless user experience for your application, but at least we are not talking about someone's life being on the line.

The problem of inferring who is whom and what is what from keys and values in your data source has been a challenge since we began writing down information about people. This problem is called *entity resolution* and has a long, storied history of technical solutions.

For any team working on entity resolution, it is important to get things right within whatever margin of error is acceptable in your business domain.

## Chapter Preview: Merging Multiple Datasets into One Graph

In this chapter, we will unveil how we merged two movie datasets, the challenges we faced along the way, and the decisions we made.

First, we will define entity resolution and how it relates to two problems we have been teaching in this book: C360 and movie recommendations.

The second section walks through the two datasets in detail. We will create a detailed understanding of the data to iteratively build up a conceptual graph model. The final graph model we build out in this section is the same conceptual graph model we introduced for development in Chapter 10.

The third section steps through our merging process. We want you to have the right expectation going into our methodology section: the type of matching and merging required with the two data sources does not need graph structure for entity resolution. We hope the details in this section help you see why.

We'll then dig into the errors we discovered during the merging process and introduce the difference between false positives and true negatives in the data.

Finally, we'll zoom back out from the specific details of merging our movie data. We will take a quick look at some common problems that misapply the use of graph structure for resolving entities in data. Then we will show a few examples in which graph structure augments an entity resolution pipeline.

Ultimately, our goal in this chapter is twofold.

First, we want to show you what it is really like to merge data. Warning: the process isn't glamorous. Merging datasets is tedious work that is often overlooked even though it is a common first step to creating a graph model.

The second goal of this chapter is to educate you on the overall problem domain. Because merging data is one of the most common first steps to creating a graph database, we want the information to help you understand all of the tools you need for solving this complex problem. Hint: most if not all of the entity resolution techniques you will most likely be using do not require graph structure to figure out who is whom.

## Defining a Different Complex Problem: Entity Resolution

The overarching theme of the matching and merging process between two data sources is a vast problem domain called *entity resolution*. Informally, the complex problem of entity resolution aims to resolve the question of who is whom or what is what across different data sources.

Is "Jon Smith" the same person as "John Smith"? Or in the case of our movie data, is a movie from MovieLens called *Das Versprechen* the same as one from Kaggle named *The Promise*?

However, in most traditional cases, a unique user identifier that links identities is likely not available for many reasons: use of external source data, unavailability of data due to user privacy constraints, or inconsistent data.

In most cases, logical identity has to be calculated from the keys and values of the present properties about each piece of data.

Historically, entity resolution (also referred to as entity matching or record linkage) relied on a set of probabilistic rules, usually defined by a domain expert, to take into account data distributions and biases of data for a particular domain. The set of probabilistic rules combine to form a functional model to calculate whether entity a is equal to entity b.

Typically, entity resolution starts with finding strong identifiers that can be linked across systems of record. After that, you begin to look for different properties about the data to determine whether or not the systems really are referring to the same logical identity.

The process for resolving identities within different data sources is outlined in Example 11-1; we will refer back to this outline a few times throughout the rest of the chapter.

*Example 11-1.*

```
A. Identify your data sources
B. Analyze the keys and values available from each source
C. Map out which keys strongly identify a single logical concept
D. Map out which keys weakly identify a single logical concept
E. Iterate until your matched and merged data is "good enough":
    1. Form a matching process
    2. Identify incorrect matches
    3. Resolve errors in the matching process
    4. Repeat Step #1
```

You analyze your sources and the keys you have within them and iteratively build up rules on how to match them together.

Sounds simple enough.

But the entire process hinges on the idea of "good enough" that we state at step E. This is where the process begins to feel more like an art than a science.

From a mathematical perspective, Figure 11-1 defines how you quantify "good enough."

$$\forall\, a, b \in D,\; f(a,b) > t \rightarrow a = b$$

*Figure 11-1. The mathematical definition of entity resolution models*

Figure 11-1 reads as: for all `a` and `b` in your dataset `D`, define a function `f`. The function `f` compares two pieces of data, `f(a,b)`, and gives you some score. If the score is above a certain threshold `t`, then we say that `a` is the same as `b`.

Example 11-1, Figure 11-1, and "Does Jon Smith = John Smith?" are all saying the same thing.

## Seeing the Complex Problem

To illustrate this complex problem, Figure 11-2 shows the concept of matching and merging data across disparate sources of data.

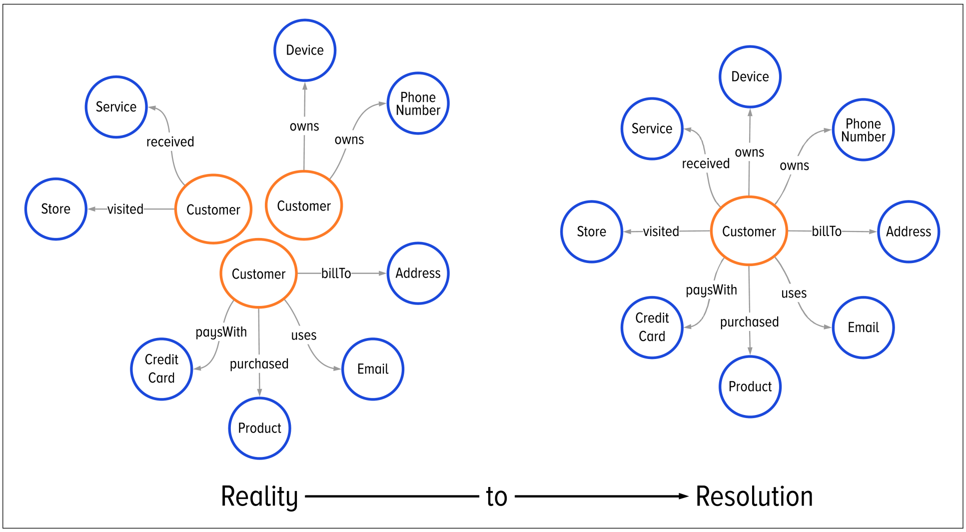



*Figure 11-2. A visual description of the conceptual problem definition for entity resolution*

Figure 11-2 displays a visualization of the entity resolution problem as a graph model. The graph on the left in Figure 11-2 illustrates the current state for most data architectures; mobile, web, and onsite databases contain disconnected views of the same customer. The most popular use of graph technology, a Customer 360 model, starts from the unified, connected graph, like what is shown on the right in Figure 11-2.

And the first example in this book started with a graph model exactly like the right side of Figure 11-2.

The ease with which you can describe all components of this problem with a graph model illustrates exactly why many teams misapply graph technology for the entire entity resolution process, most of which doesn't rely on a graph for its technical solution.

So let's put entity resolution into practice.

# Analyzing the Two Movie Datasets

We want to show you how we analyzed two popular open source movie datasets—MovieLens and Kaggle.[1] Our process parallels steps A through D in Example 11-1.

We selected the MovieLens dataset so that we could use a very diverse and well-documented dataset of user ratings of movies. The Kaggle dataset augments the Movielens data with details and actors for each movie.

> Deciding to bring together two datasets ended up being one of the best decisions we made for this book because it required us to really dig into the process of what it is like to get started with graph technology. To illustrate that, this section walks you through exactly how we reasoned about our conceptual graph data model as we merged these two datasets.

Starting with the MovieLens source, we will take a look at the datafiles available and how they will fit together. Then we will walk through the Kaggle data. The most important part of this process is identifying which keys and values from the Kaggle dataset refer to the same logical concepts from the MovieLens dataset. Specifically, we will be looking for which strong identifiers we can use to match the datasets together.

The upcoming section is long and detailed to give you a real glimpse into the process.

## MovieLens Dataset

There are six files that we used for our schema and example from the MovieLens dataset:

1. `links.csv`
2. `movies.csv`
3. `ratings.csv`
4. `tags.csv`
5. `genome-tags.csv`
6. `genome-scores.csv`

1 F. Maxwell Harper and Joseph A. Konstan, "The MovieLens Datasets: History and Context," *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, no. 4 (2016): 19, *https://doi.org/10.1145/2827872*.; Rappeneau, Stephane. "350 000+ movies from themoviedb.org," Kaggle, 19 July 2016, *https://www.kaggle.com/stephanerappeneau/350-000-movies-from-themoviedborg*.

We are going to step through each of the six files while we iteratively construct our developmental graph model from Chapter 10. There are only five upcoming subsections, however, because we are going to talk about `genome-tags.csv` and `genome-scores.csv` in the same section.

### 1) Links

We started with the `links.csv` file from MovieLens because it is the source of strong identfiers to external data sources. The `links.csv` file contains 27,278 lines of linking identifiers that can be used to link external sources of movie data. Each line of this file after the header row represents one movie and has the following format:

```
movieId,imdbId,tmdbId
```

Each strong identifier is defined as follows:

1. `movieId` is an identifier for movies used by the MovieLens project.
2. `imdbId` is an identifier for movies used by IMDB.
3. `tmdbId` is an identifier for movies used by TMDB.

For example, the movie *Toy Story* has a `movieId` of 1 (*https://movielens.org/movies/1*), an `imdbId` of `tt0114709` (*http://www.imdb.com/title/tt0114709*), and a `tmdbId` of 862 (*https://www.themoviedb.org/movie/862*).

We started the data modeling process with this file and built the schema shown in Figure 11-3.

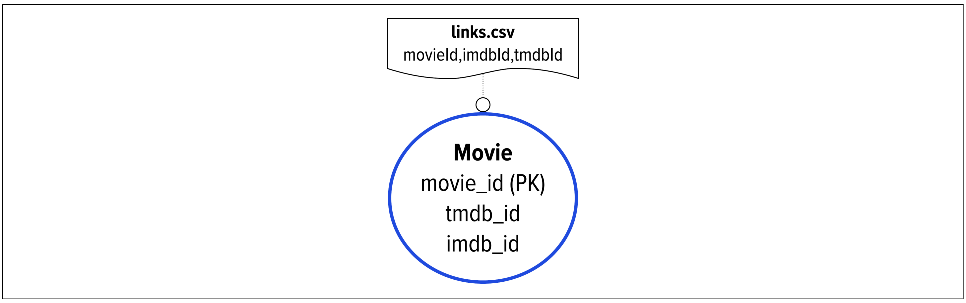



*Figure 11-3. The first step in our data modeling process with the MovieLens dataset: mapping the values from the links file into a vertex label*

The schema in Figure 11-3 has one vertex label: `Movie`. This vertex has a partition key of the `movie_id` and two additional properties: `tmdb_id` and `imdb_id`. We changed the casing from `camelCase` to `snake_case` to conform to naming standards when working with Apache Cassandra.

Following the process we outlined in Example 11-1, we learned the following information about this file:

1. There are 27,278 movies in total.
2. 27,278 movies have an `imdbId` (100% coverage).
3. 26,992 movies have a `tmdbId` (98.95% coverage). Note: these movies also have an `imdbId`.
4. 252 movies are missing a `tmdbID`.

Those of you who are checking the math here may have observed that 27,278 does not equal 26,992 + 252. The comparison is off by 34 because there are 17 errors in this dataset in mapping a movie's `tmdbId` to its `imdbId`. We will delve into this issue in a later section.

This information tells us that the MovieLens data has 100% coverage of strong identifiers from the IMDB data source. Therefore, the first identifier to check when matching into this data will be the `imdb_id`.

Let's look at the dataset that starts to populate the data model with more information about each movie.

### 2) Movies

The MovieLens dataset has a `movies.csv` file that contains a title and genres for each movie. The MovieLens resources indicate that we connect this data to the `links.csv` information via the `movieId`.

There is an entry in the movies file for each of the 27,278 movies in the dataset. Each line has the structure:

```
movieId,title,genre_1|genre_2|...|genre_n
```

According to the MovieLens documentation, the movie titles were entered manually or imported from the MovieLens project. The genres are a pipe-delimited list and are selected from topics such as Action, Adventure, Comedy, Crime, Drama, and Western.

We discovered that there are 18 unique genres in this set.

We continued the data modeling process with this file and added the schema. The next iteration of our schema is shown in Figure 11-4.

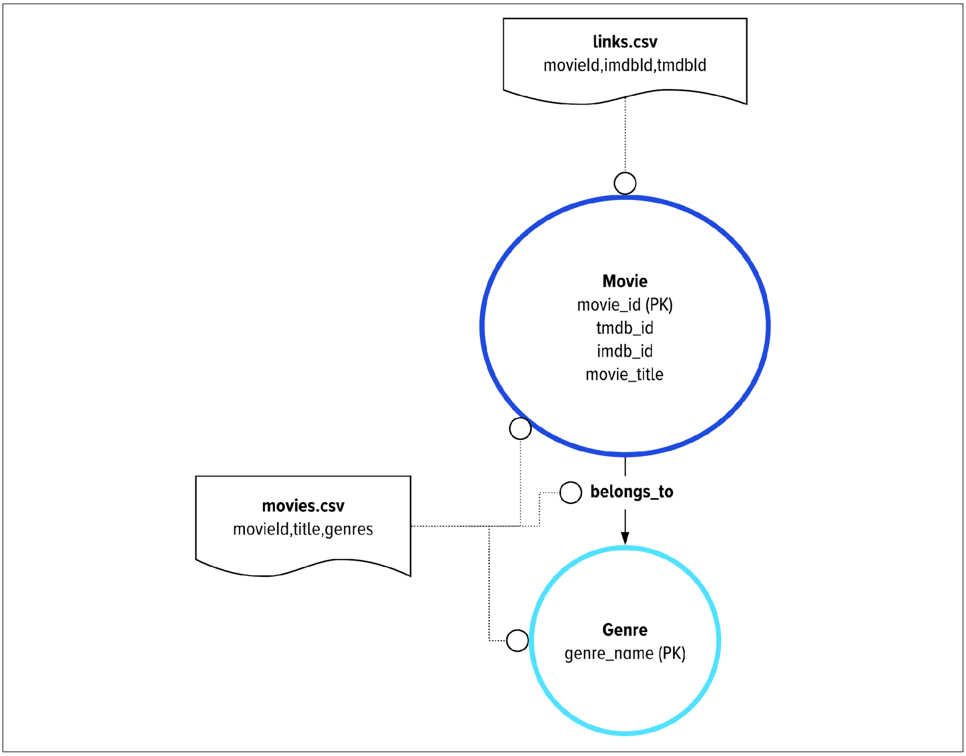



*Figure 11-4. The second step in our data modeling process with the MovieLens dataset: mapping the values from the movies file into an edge, a new vertex label, and new properties*

The `movies.csv` file gave us three additions to our data model. The mapping of the `movies.csv` file to three additions to the data model is shown in Figure 11-4.

First, we augmented the `Movie` vertex to have a `movie_ title` property. Second, we created a `Genre` vertex and partitioned that vertex by the `genre_name`. Third, we created an edge from the `Movie` vertex to the `Genre` vertex with the label `belongs_to`.

We needed the MovieLens dataset for its user ratings. Let's take a look at that file next.

### 3) Ratings

There are 20,000,263 ratings from users to a movie in the `ratings.csv` file. Each line of this file represents one rating by one user. The format of the file is:

```
userId,movieId,rating,timestamp
```

This file gives us our first glimpse of users from the MovieLens database. There are 138,493 unique `userIds` across the 20-million-plus ratings. Ratings are made on a

5-star scale, with half-star increments [0.5 stars, 5.0 stars]. Timestamps are in epoch: seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

The `ratings.csv` file introduces a new vertex and edge label into our data model. Figure 11-5 augments the schema with this new information.

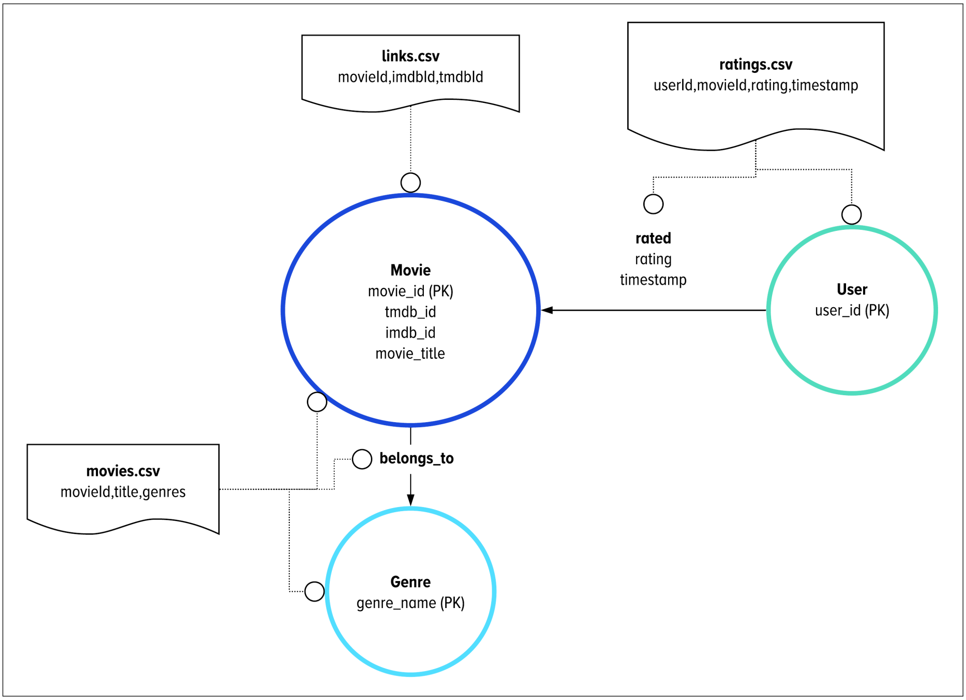



*Figure 11-5. The third step in our data modeling process with the MovieLens dataset: mapping the values from the ratings file into vertex and edge labels*

As seen in Figure 11-5, our data model now has a `User` vertex label and a `rated` edge label. We partitioned the `User` vertices by the `user_id`. We added the `rating` and `timestamp` properties onto the `rated` edge.

### 4) Tags

In addition to ratings, the users also provided their own tags about the data. Each tag is a single word or short phrase and was created by the user. There are 465,564 tags created by users about movies.

Each line in the `tags.csv` file has the structure:

```
userId,movieId,tag,timestamp
```

We use the information from the tags file to continue to build our data model. The next iteration is shown in Figure 11-6.

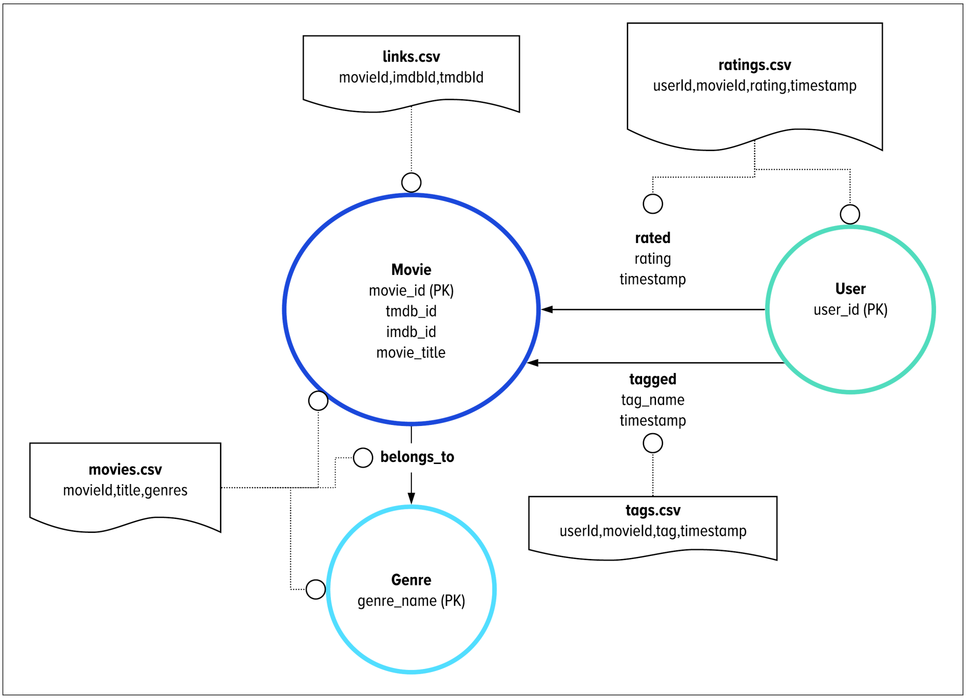



*Figure 11-6. The fourth step in our data modeling process with the MovieLens dataset: mapping the values from the tags file into vertex and edge labels*

As illustrated in Figure 11-6, we can link a tag from a user to a movie using the `userId` and `movieId`. We modeled the `tag_name` and `timestamp` on the `tagged` edge.

There is one last concept from the MovieLens data to add to our data model: the tag genome.

### 5) Tag genome

There are two files that you can find within the collection of MovieLens datasets: `genome-tags.csv` and `genome-scores.csv`. These two files analyze the tags we modeled in Figure 11-6 and represent how strongly a movie can be described by properties from the user tags.

The tag genome was computed using a machine learning algorithm on user-contributed content, including tags, ratings, and textual reviews.[2]

2 Jesse Vig, Shilad Sen, and John Riedl, "The Tag Genome: Encoding Community Knowledge to Support Novel Interaction," *ACM Transactions on Interactive Intelligent Systems (TiiS)* 2, no. 3 (2012): 13, *http://doi.acm.org/10.1145/2362394.2362395*.

The file `genome-scores.csv` contains 11,709,768 movie-tag relevance scores in the following format:

```
movieId,tagId,relevance
```

The second file, `genome-tags.csv`, provides the tag descriptions for 1,128 tags in the genome file, in the following format:

```
tagId,tag
```

The tags give us a new vertex label and edge label for this data set, and are the last iteration in modeling with the MovieLens data. The `tagId` will map to the partition key for the `Tag` vertex, `tag_id`, and the `tag` will map to `tag_name`. Let's take a look in Figure 11-7.

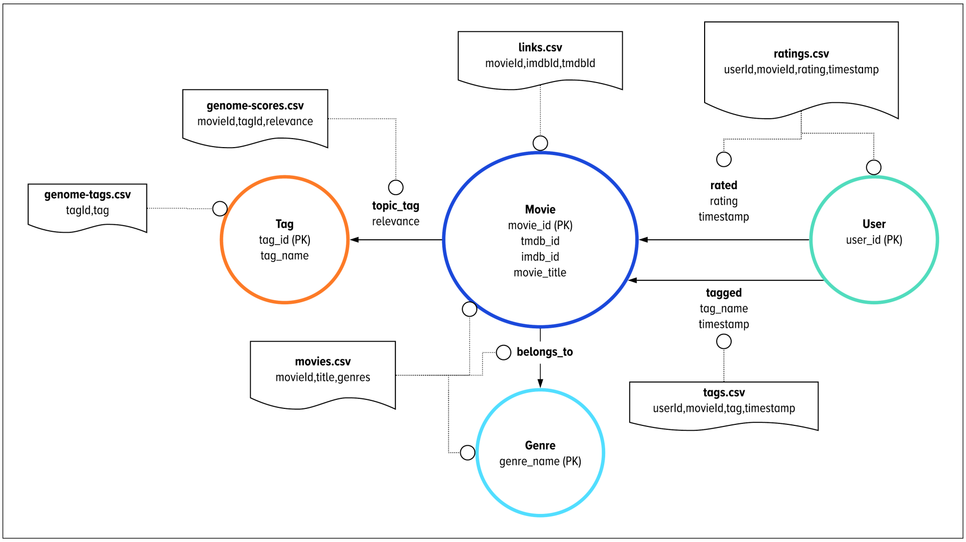



*Figure 11-7. The last step in our data modeling process with the MovieLens dataset: mapping the values from the genome files into our schema*

The conceptual data model in Figure 11-7 represents the full mapping of the MovieLens data into a graph model. The strong identifiers within this model are the most important pieces to understand and follow. Among all of the strong identifiers, the most important to follow is `movie_id`, because it is used in every file to connect each concept to a movie.

You may choose to map the data differently, and that is OK. It all comes down to how you end up querying the information and the questions you want to ask from these sets in a production environment.

The model we have in Figure 11-7 is a good starting place for development.

Let's build on this model with the data available from Kaggle.

## Kaggle Dataset

There are two main sources of information from a dataset on Kaggle that we are going to use to augment our data model: movie data and actor data. Let's follow the same process as we did with the MovieLens data to continue to build our data model.

### Movie details

The Kaggle dataset is an excellent source for two reasons. First, it contains the most complete listing of movie information, with data available for 329,044 unique movies.

The plethora of details available for each movie is the second reason the Kaggle data is an excellent source. The file that contains all of the details about a movie is `AllMoviesDetailsCleaned.csv`. There are 22 different headers in this file that describe additional publicly available information about a movie, such as its budget, original language, overview, popularity, production companies, runtime, tagline, release date, and many other facts.

The most important keys in this data are `id` and `imdb_id`. Here is what we learned about the strong identifiers from the Kaggle data:

1. The `id` from the Kaggle dataset maps to the `tmdb_id` from TMDB.
2. The `imdb_id` maps to the movie IDs from IMDB.
3. All 329,044 movies from the Kaggle dataset have identifiers from TMDB.
4. 78,480 movies from the Kaggle dataset are missing an ID from IMDB.
5. The only other information we have from Kaggle to compare with the MovieLens data is a movie's title.

The coverage of strong identifiers in the Kaggle dataset helps us begin to understand how we are going to match and merge this data with MovieLens. The Kaggle data source has 100% coverage on strong identifiers from TMDB, whereas the MovieLens data source has almost 100% coverage on strong identifiers from IMDB.

A mismatch in strong identifier coverage between the data sources is both bad and good. It is bad because the matching process is not going to be straightforward. The silver lining, however, is that this example is going to make for a great educational tool on matching data.

From the `AllMoviesDetailsCleaned.csv` file, we pulled seven pieces of information to augment our data model. Figure 11-8 illustrates the next stage in the development of our data model.

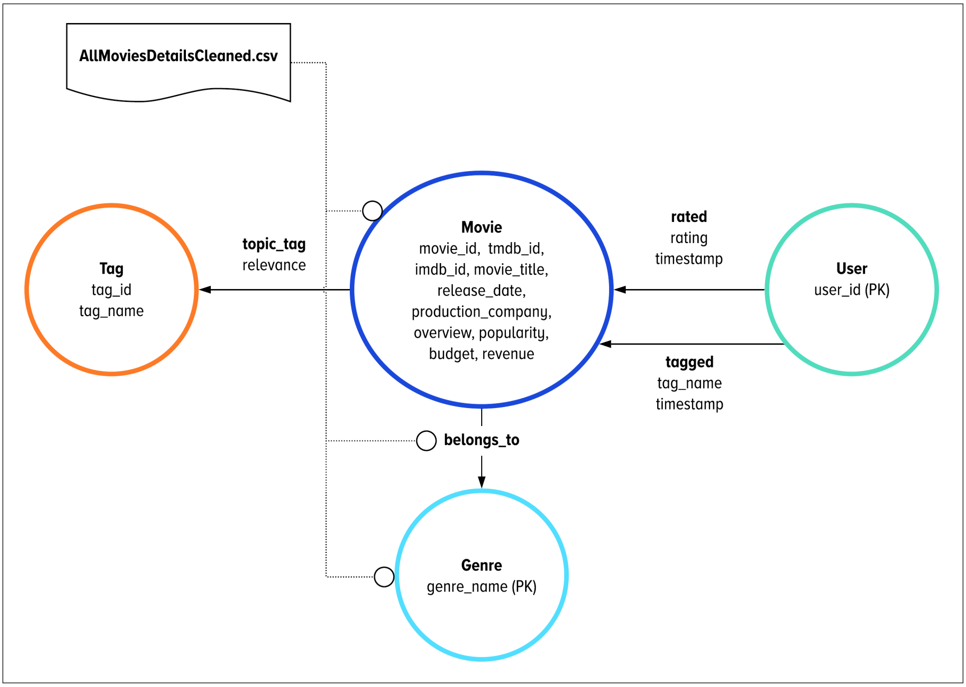



*Figure 11-8. The first of two steps in augmenting the MovieLens data with the Kaggle dataset: adding properties to the movie vertices*

Figure 11-8 shows six new properties we added to the movie vertex: release date, production company, overview, popularity, budget, and revenue. The seventh detail we pulled from the Kaggle data was the genre property. This augmented more `Genre` vertices and edges from movies to genres.

We needed this dataset so that we could merge in the information about actors for each movie. Let's look at how we can access that information.

### Actors and casting details

The file `AllMoviesCastingRaw.csv` provided information about actors, directors, producers, and editors for each movie. We selected only the actors to include in our examples.

The `AllMoviesCastingRaw.csv` file lists five actors for each of the 329,044 movies. This information is listed on one line, with the following structure for the first 11 columns:

```
id,actor1_name,actor1_gender, ..., actor5_name, actor5_gender....
```

Each actor was connected to their movie by matching the ID to the `tmdb_id` of the `Movie` vertex.

Additionally, we created `collaborator` edges for actors who were in the same movie. We used the `release_date` from the `AllMoviesDetailsCleaned.csv` file to add a year to each of these new edge labels about actors.

Figure 11-9 shows the data model we arrived at for our example.

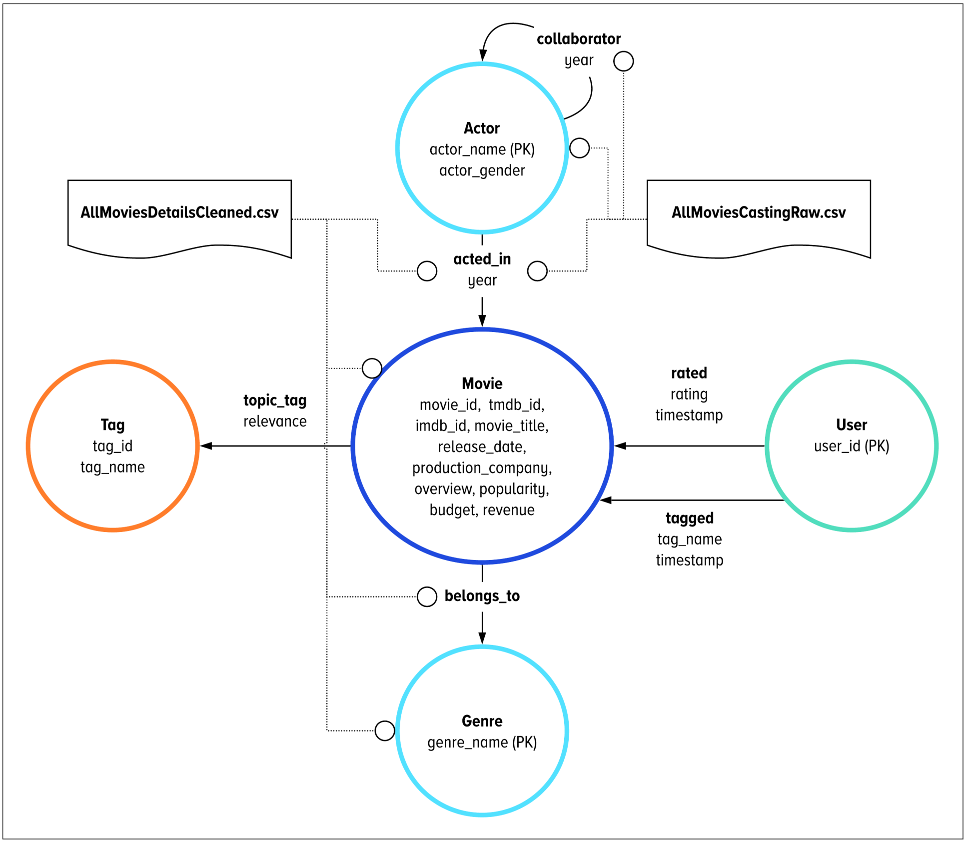



*Figure 11-9. The second of two steps in augmenting the MovieLens data with the Kaggle dataset: adding actors into the model*

> Figure 11-9 shows the merged data model with the MovieLens and Kaggle datasets. We didn't include everything available from the Kaggle dataset. If there is something you would like to use, please visit us at *https://oreil.ly/graph-book*. We will accept pull requests for the data and processes that accompany this text.

## Development Schema

The data integration process between the MovieLens and Kaggle sources created the development schema we will be using in our examples. Using the Graph Schema Language (GSL), the development schema is shown in Figure 11-10.

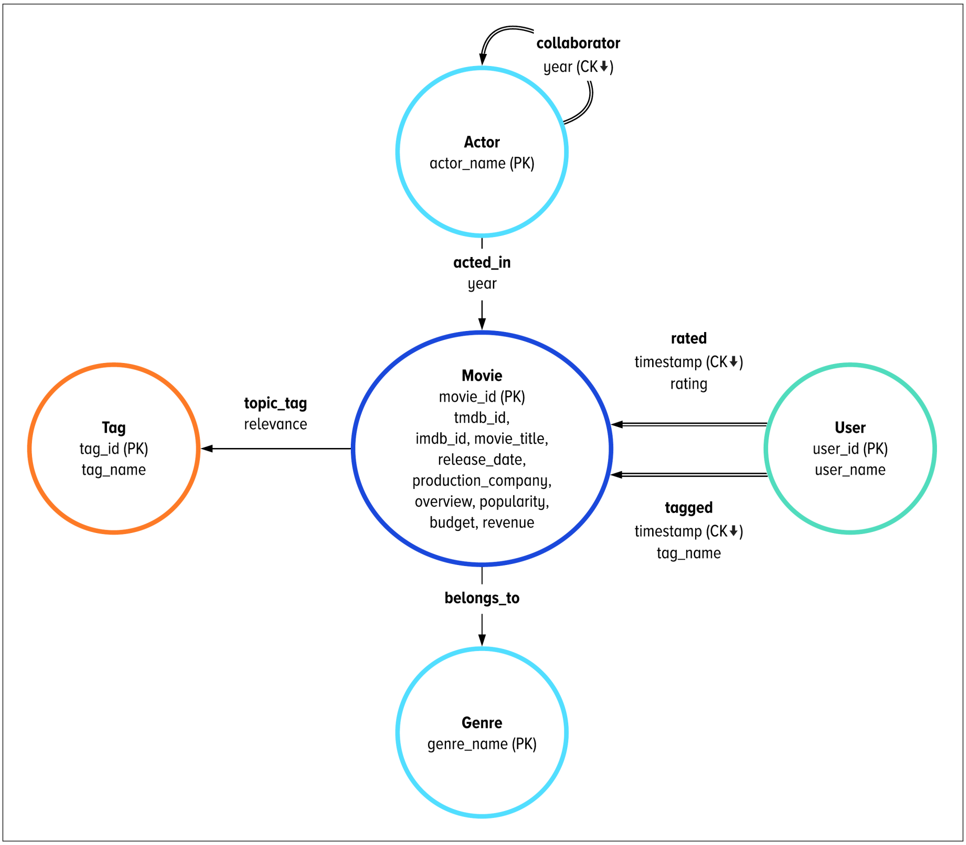



*Figure 11-10. The development schema for our merged database about movies*

Chapter 10 showed you how to use the GSL to translate Figure 11-10 into schema statements. We designed this process for you to follow the same idea popularized by ERDs.

## Matching and Merging the Movie Data

The idea to merge the MovieLens and Kaggle datasets for this example became a much more difficult and involved task than we anticipated.

And in our experience, problems in matching and merging data sources are always more involved than you anticipate.

The process of resolving two data sources starts with mapping the strong identifiers present in both systems for linking the data. We just went through that in the last section. We learned that the strong identifiers available across both datasets are the movie identifiers from TMDB and IMDB.

However, each dataset has a different distribution of these IDs. After studying the two sources, we learned the following:

1. Each entry in the MovieLens dataset has an IMDB identifier.
2. 1% of the movies in the MovieLens dataset are missing a TMDB identifier.
3. Each entry in the Kaggle dataset has a TMDB identifier.
4. 24% of the movies in the Kaggle dataset are missing IMDB identifiers.

From this information, we know that we are going to have to build a process that uses both IDs from the datasets because we can't always rely on either.

> When you need to merge data sources, always start with finding and understanding the distribution of strong identifiers in each system!

Let's discuss how we procedurally matched and merged the data between the sources when everything matched up correctly. After this next section, we will walk through the errors we discovered in both datasets and how we resolved them.

### Our Matching Process

At the start, our merging process was simple. We started by processing the MovieLens data. Then we had to figure out what it meant to be a match from the Kaggle dataset and how to merge the information.

We defined a match from the Kaggle data source into the MovieLens data source as being when there was exactly one entry in the MovieLens dataset that matched on one or both of the TMDB and IMDB identifiers.

The steps we followed for a successful match and merge are what we have in Example 11-2.

*Example 11-2.*

```
1 For each movie_k in the Kaggle dataset:
2     movie_m = MATCH MovieLens data by the tmdb_id of movie_k:
3     if there is a movie_m:
4         if imdb_id of movie_k == imdb_id of movie_m:
5             movie_m2 = MATCH MovieLens data by the imdb_id of movie_k:
6             if tmdb_id == tmdb_id_m2:
7                 UPSERT the kaggle data
8     else:
9         movie_m = MATCH MovieLens data by the imdb_id of movie_k:
10            if movie_m is not null:
11                if imdb_id identifiers match:
12                    UPSERT the data
13            else:
14                We know movie_k is not in the MovieLens data
15                INSERT movie_k from Kaggle
```

The process in Example 11-2 matched 26,853 movies that were in both databases. Before the matching process, there were 252 movies in the MovieLens database with no IMDB identifier; 15 of those movies were found and resolved with the Kaggle dataset according to their TMDB identifier.

> You may be wondering why the logic on lines 5 and 6 of Example 11-2 is necessary. It turns out there were some errors in the source data. We will get into those errors in "Resolving False Positives" on page 343.

For a deeper example, Figure 11-11 illustrates how the movie *Toy Story* would be successfully matched between the two datasets using the process outlined in Example 11-2.

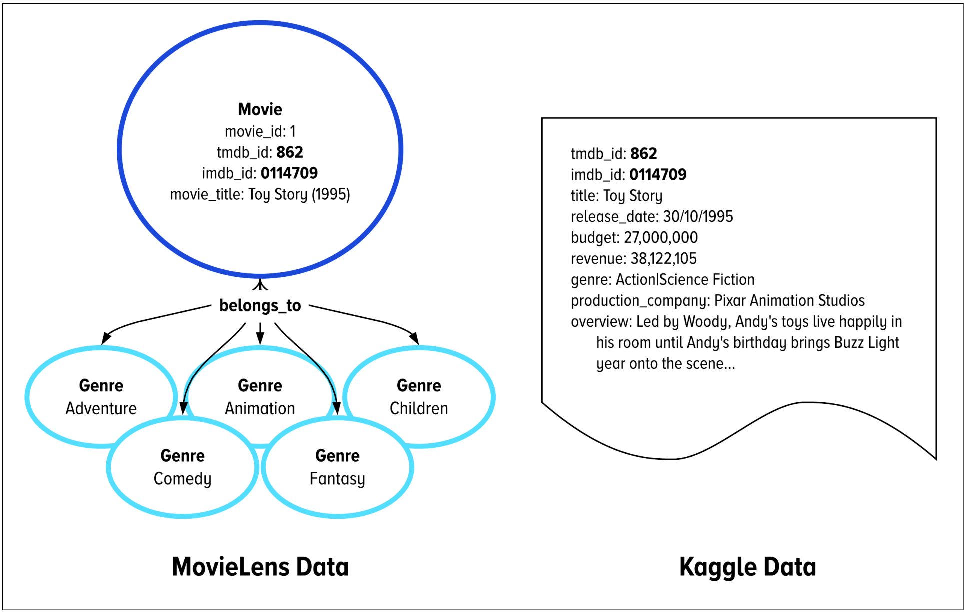



*Figure 11-11. Showing how the Kaggle data source would be matched with the MovieLens data source for the movie Toy Story*

The most important feature of Figure 11-11 is to observe the values for the strong identifiers between the two sources. We already modeled a movie titled "Toy Story (1995)" from the MovieLens data with a `tmdb_id` of 862 and an `imdb_id` of 0114709. When we processed the Kaggle movie, the algorithm worked as shown in Example 11-3.

*Example 11-3.*

```
For the "Toy Story" movie in the Kaggle dataset:
    movie_m = search MovieLens data by the 862:
    if there is a movie_m:
        if 0114709 == 0114709:
            movie_m2 = search MovieLens data by the 0114709 of movie_k:
            if 862 == 862:
                UPSERT the kaggle data
```

We used UPSERT when we inserted the data because the underlying datastore is Apache Cassandra. In this and most situations, UPSERTs are the fastest way to handle writes.

Figure 11-12 shows the merged version of the *Toy Story* movie that ended up in our dataset.

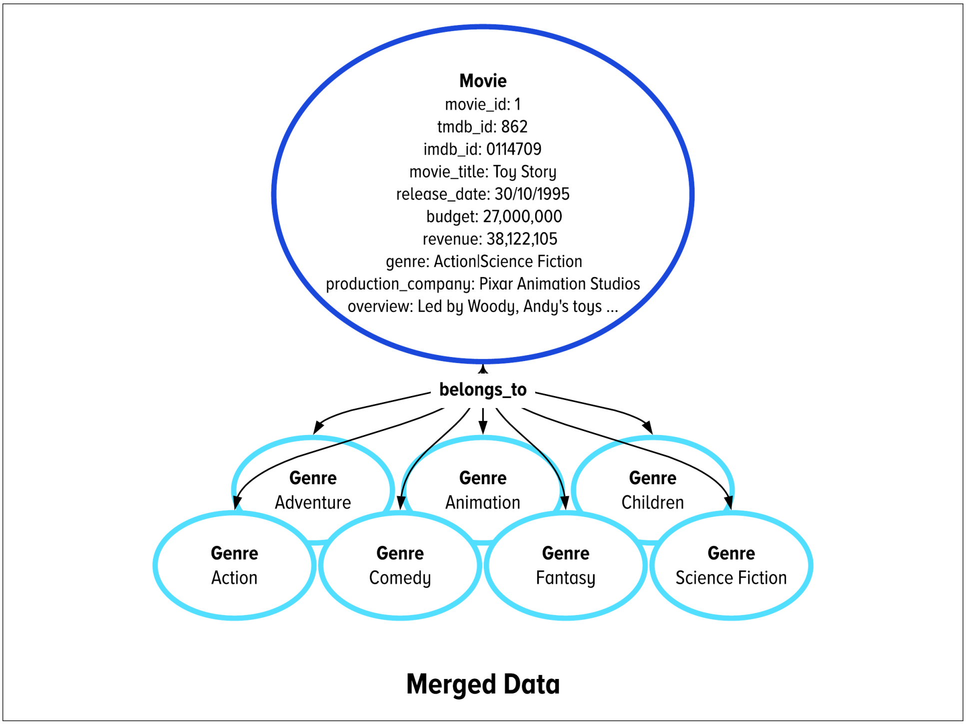



*Figure 11-12. The final view of the Toy Story movie after the merged data*

Along the way, we documented the tripping points and decisions we had to make. We are going to walk through those in the next section.

# Resolving False Positives

When you first read through the matching process in Example 11-2, you may have thought that some of the additional checks were redundant. For example, when determining whether the Kaggle data matches a MovieLens movie, we first found a movie by its TMDB identifier and then looked for it again by its IMDB identifier. Only when all of these scenarios found the same movie and identifiers did we consider it a match.

However, the process we started with discovered something really interesting about the MovieLens data: it contained false positives within its own data.

## False Positives Found in the MovieLens Dataset

The matching process outlined in Example 11-2 first revealed errors within the links from the MovieLens database. Specifically, there were 17 occurrences in the

MovieLens data of the same TMDB identifier pointing to different IMDB identifiers. This is referred to as a *false positive*.

*False positive*
: A false positive error occurs when the entity resolution process links two references that are not the same.

We discovered the false positives within the MovieLens data when we were trying to merge a Kaggle record based on its TMDB identifier. When a Kaggle entry matched a MovieLens movie by their respective `tmdb_ids`, the sequential lookup by the Kaggle entry's IMDB identifier returned two results from the MovieLens data.

Let's look at some of the false positives that exist within the MovieLens data (see Table 11-1):

*Table 11-1. Showing six false positives due to the incorrect mapping of `tmdbId` to `imdbId` in the MovieLens data*

| movie_id | imdb_id | tmdb_id | movie_title |
|---|---|---|---|
| 1533 | 0117398 | 105045 | The Promise (1996) |
| 690 | 0111613 | 105045 | Das Versprechen (1994) |
| 7587 | 0062229 | 5511 | Samouraï, Le (Godson, The) (1967) |
| 27136 | 0165303 | 5511 | The Godson (1998) |
| 8795 | 0275083 | 23305 | The Warrior (2001) |
| 27528 | 0295682 | 23305 | The Warrior (2001) |

To know whether these are the same or different movies requires crawling the original sources. We didn't do that work for these examples. Therefore, for these 17 instances of clashing mappings within the MovieLens data, we removed both of each pair of records, or a total of 34 instances, from the MovieLens source.

From deeper research on IMDB and TMDB, we found that the Kaggle dataset had the correct entries. Therefore, we used the Kaggle data as the ground truth in these instances.

After resolving the issue within the MovieLens source, we collected information about the errors found when mapping the two data sources together.

## Additional Errors Discovered in the Entity Resolution Process

Some statistics about the errors and incorrect matches we found between the datasets are:

1. Zero movies had matching TMDB identifiers but mismatched IMDB identifiers.
2. Merging the datasets produced 143 errors in which the sources had matching IMDB identifiers but mismatched TMDB identifiers.

At the start, we did not know whether these 143 errors were false positives or false negatives. We needed to examine them to figure out what type of error they represented.

The additional data about the 143 mismatched movies that is available for comparison is as follows:

1. The movie's title in each database
2. The public page about the movie on IMDB
3. The public page about the movie on TMDB

When resolving errors, you want to start with the data you have. In this case, we can compare movie titles. The breakdown of how those titles compared is shared in Table 11-2.

*Table 11-2. Diving into the reasons that movie titles mismatched between the MovieLens and Kaggle datasets*

| Reasons the titles were different | Total Occurrences | Percentage |
|---|---|---|
| "A" | 5 | 3.50% |
| Actually different | 9 | 6.29% |
| Same, but different languages | 1 | 0.70% |
| "The" | 36 | 25.17% |
| (year) | 92 | 64.34% |

The MovieLens data source indicates that MovieLens augmented the titles of movies to contain the release year when that information was available. Therefore, we would expect to see that many of the clashes in titles are due to how this data was prepared. Table 11-2 confirms this, with 64% of the mismatched movies between the two sources having titles where the MovieLens title has the `(year)` but the Kaggle title does not.

The remaining reasons the titles were different between the MovieLens and Kaggle datasets are fairly interesting:

1. 3.5% of the time, one title had the word "A" in it, whereas the other title did not.
2. 25.1% of the time, one title had the word "The" in it, whereas the other did not.

3. There was one occurrence in which the titles were the same but in different languages: *The Promise* (English) versus *La Promesse* (French).
4. There were nine occurrences of the two titles actually being different.

The analysis of differing titles did not go far enough to say whether or not the movies were the same.

For 10% of these mismatched movies, which is 15 movies, we looked at their movie details in TMDB and IMDB to see which source had the correct information. From this deeper analysis, we found that the Kaggle data source had the correct TMDB and IMDB identifiers in all of the cases we investigated. The details of our in-depth study of mismatched movies are:

1. In 12 out of the 15 cases, the MovieLens data contained a TMDB identifier that pointed to a web page that had been removed.
2. 15 of the 143 incorrectly matched movies had the correct information in the Kaggle data source based on crawling the original sources at TMDB and IMDB.
3. For all of the incorrect mappings that we deeply investigated, the MovieLens data source never had the correct information.

As a result, for all of the 143 occurrences in which the strong identifiers did not match up, we relied on the information from the Kaggle data source. That is, our final resolved errors contained 143 more false positives where the MovieLens data incorrectly linked a TMDB identifier to an IMDB identifier.

## Final Analysis of the Merging Process

After we finished the resolution process, there was a total of 329,469 movies in our merged database. Some additional statistics about the merged dataset are:

1. There are 26,853 movies that are in both the MovieLens and Kaggle data sources.
2. There are 78,480 movies in our merged database with no IMDB identifier.
3. There are 237 movies in our merged database with no TMDB identifier.

We hope you found the details on how we merged these datasets to be illustrative and representative of the not-so-glamorous process of merging datasets. It is a common first step that every team has to go through before it can get started with using its data in a graph.

Which does raise the question: how could a graph help resolve our movie data?

## The Role of Graph Structure in Merging Movie Data

While we are talking about resolving the false positives in the movie data, there is one area in which we could use edges in our data to resolve some of the false positives. Let's take a look at a specific case.

If we had the actors from the MovieLens source, we could (hypothetically) use graph structure to help resolve some of our false positives. For instance, consider the two movies listed in Table 11-3 that are false positives from the MovieLens data.

*Table 11-3. Example movies that require additional data to determine whether they are the same movie*

| movie_id | imdb_id | tmdb_id | movie_title |
|---|---|---|---|
| 8795 | 0275083 | 23305 | The Warrior (2001) |
| 27528 | 0295682 | 23305 | The Warrior (2001) |

Table 11-3 shows all of the information that we have about these two movies. And from the data we have, we cannot confidently conclude whether these are or are not the same movie. The TMDB identifiers are the same, but the IMDB identifiers are different. However, the titles are identical.

The data we have just isn't enough to make a conclusive decision. So let's see what we can figure out about these two movies so we can come to a conclusion.

After doing some deeper digging, we could use the IMDB data to get the actors for each of these movies. Given the actor information about each movie, Figure 11-13 displays what each of their graphs would look like:

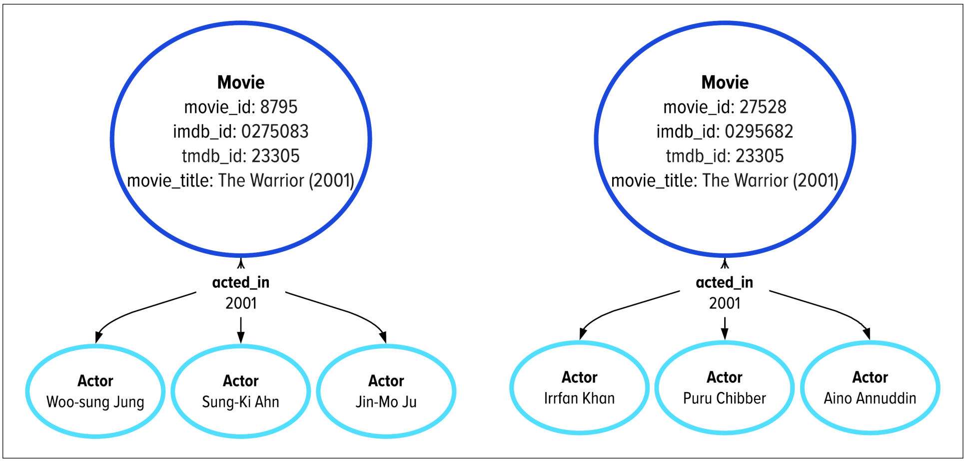



*Figure 11-13. A view of the two movies titled The Warrior (2001) after deeper research on the actors for each of these movies*

By resolving actors and creating relationships from movies to their actors, we can see that these movies are actually distinct and different movies. They have no actors in common between their cast lists (though we are showing just the first three actors in each cast list in Figure 11-13).

Figure 11-13 gives you an idea of when using edges from a graph can help you discover whether or not the data you have is distinct.

The lesson of the simple entity resolution example in this chapter is that the majority of your tasks in entity resolution do not require a graph structure. Well-defined processes start with following exact matches of strong identifiers. In cases when strong identifiers are not enough, you can rely on character edit distances for the next most important keys and values about your data.

Then, after you have covered the basics, and if relationships make sense in your data, you may want to bring relationships into your entity resolution process.

Figure 11-13 illustrates a compelling reason to use graph structure for entity resolution in our example (after we resolved strong identifiers and names with edit distances!) because you can immediately see that the movies are different. And you can infer why they are different. Although we certainly can't do this kind of analysis for all problems, a graph can be a far more useful tool to add into your entity resolution process than digging deeper into tabular information to sort out the answer.

The ability to use a graph to resolve and merge data is a multifaceted problem. Elaborating on the full details of where, when, and how to use a graph for generalized entity resolution would fill a whole book.

From here, let's get back to how we can deliver these recommendations at scale within a production application.

# CHAPTER 12
# Recommendations in Production

Pretty much every application you use these days has a "recommended for you" section.

Just think about your favorite applications for digital media, apparel, or retail providers. We rely on the recommendation pane in our media apps to find new movies to watch or books to read. Brands like Nike tailor your in-app experience with personal and customized wardrobes. Even your local grocery store's app delivers recommended coupons to you for your next visit.

Recommendations and personalization have infiltrated almost every nook and cranny of our digital experience.

But how do you build a process that delivers recommendations within an application at the speed that we have all learned to expect?

As we walked through in Chapter 10, it is very possible to connect data sources with a graph and create personalized recommendations for a user. However, the sheer amount of data that is required to process a graph-based recommendation at scale significantly limits how you would use collaborative filtering within a production application.

We don't think a user of Nike's apparel app is going to wait the multiple seconds required to process an end-to-end NPS-inspired collaborative-filtering graph query. And neither should you.

Instead, we encourage you to think like a production engineer. We want to set up procedures that prioritize the end user's in-app experience and then figure out how to connect a longer running query, like a graph-based collaborative filter, with a process that can guarantee your end user receives recommended content within web response time.

The focus of this chapter is just that: teaching you how to break down a complex graph problem into a piece that can be queried in real time versus a piece that requires a batch process.

## Chapter Preview: Understanding Shortcut Edges, Precomputation, and Advanced Pruning Techniques

There are four main sections to this final chapter.

We will start by explaining shortcut edges. We will show you why our development process doesn't scale and how shortcut edges solve our problem. We also will talk about different ways to use shortcut edges with your data with different pruning techniques.

In the next section we'll explain how we precomputed shortcut edges for our movie data. We will be diving into data parallelism and the different operational challenges you will face when integrating longer running calculations to be used in a transactional query.

Our third section will introduce the final production schema we used for our movie data. We will walk through the schema code and how to load the edges we computed, as you have done many times already.

In the last section we will show you how to use the shortcut edges to deliver recommendations to your end users. We will dig deeply into the partitioning strategies within Apache Cassandra so that you can reason about the latencies for different types of recommendation queries with our data.

## Shortcut Edges for Recommendations in Real Time

We left off our discussion of recommendations in Chapter 10 with a graph query that performed collaborative filtering on our graph data. We created and computed an NPS-inspired metric to figure out which movie we should recommend according to the movies rated by one of our users. Figure 12-1 illustrates the general concept behind the approach we built.

Figure 12-1 shows how we walked through our development graph data from the left to the right to find recommendations. If you were following along in the notebook, you likely noticed that the overall processing time for these queries is not going to cut it if you want to use this approach in a production application. The user will end up waiting way too long to get their recommendations because the query takes too long to process.

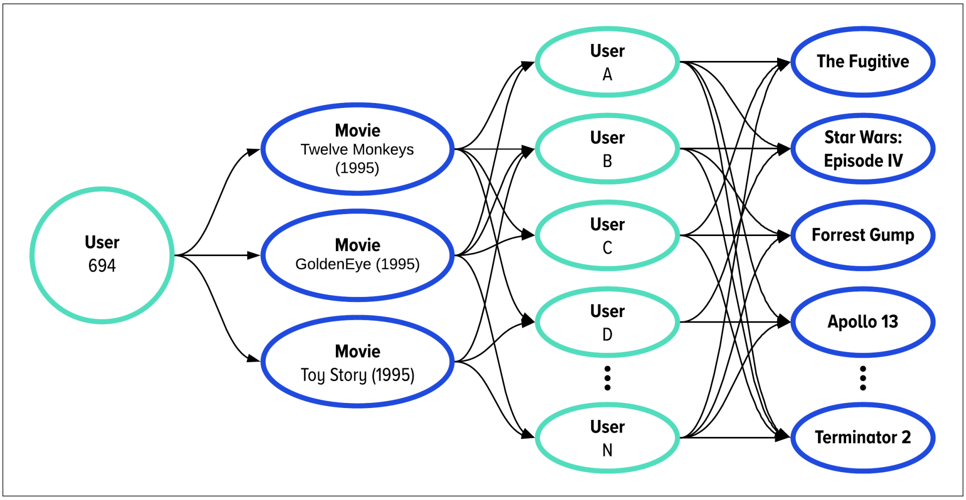



*Figure 12-1. An example of where we left off to highlight the sheer volume of data required to process our development queries from Chapter 10*

Let's dig into why and then how we will resolve the issues.

## Where Our Development Process Doesn't Scale

The reasons our development graph queries won't scale are simple to state: branching factor and supernodes. If you think like us, you'll agree that the appropriate response to having to deal with both of these problems at the same time is a very sarcastic "great."

However, if you recall, we have run into issues with your graph's branching factor and supernodes before.

We first ran into branching factor in Chapter 6 within our sensor network when we tried to walk from a tower down to all sensors. The branching factor of the edges in our data created exponential growth in our processing overhead.

The same branching problem exists within the general class of recommendation problems. As you walk from a user to movies to users to movies, your queries fork an exponential number of traversers in order to process all of the edges within the data.

We also have to deal with supernodes in our collaborative-filtering queries. Supernodes are very closely related to branching factor: supernodes represent the extreme end of your graph's branching factor, as they are the highest degree vertices.

We first experienced supernodes in Chapter 9 as we created filters and optimizations for pathfinding. We specifically eliminated high degree vertices from our pathfinding

queries because they (usually) do not provide meaningful results in pathfinding applications.

We are going to have to deal with supernodes differently in our recommendation data.

In recommendation problems, we have two types of supernodes: the superuser and the superpopular content. A superuser is a member of your platform who has viewed or rated almost every piece of content. Any time that user is discovered during your collaborative-filtering query, a large number of movies are inserted into your result set. There is also very popular content that is viewed or rated by most users on your platform.

Unlike how we dealt with these supernodes in pathfinding, in a recommendation system you want to account for this type of popularity in your algorithms because it can indicate trending or highly probable recommendations.

So how do we get around both of these problems? We build connections around them.

## How We Fix Scaling Issues: Shortcut Edges

We have one last production trick to teach you: the shortcut edge. Shortcut edges are one of the most popular tricks used by teams around the world to mitigate the combined risk of your graph's branching factor and supernodes in production queries.

*Shortcut edge*
: A shortcut edge contains precomputed results of a multihop query from vertex `a` to vertex `n` to be stored as an edge directly from `a` to `n`.

Let's look at how we will be using shortcut edges in our example in this chapter. Figure 12-2 shows how we will use an edge called `recommend` to directly connect movies to their recommendations according to the NPS-inspired metric of our user ratings in the middle.

The `recommend` edge is essentially building a bridge over the riskiest part of our collaborative-filtering query to ensure that the end user in our application does not have to wait.

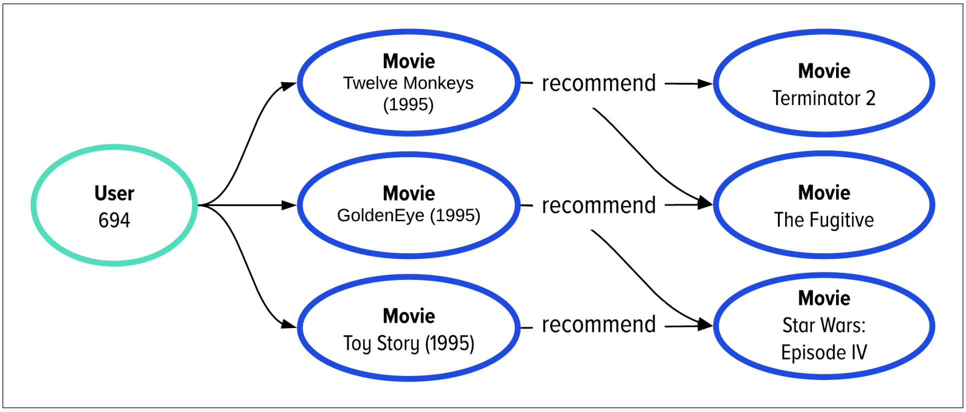



*Figure 12-2. An example of using a precomputed `recommend` edge as a shortcut from a movie directly to its recommendation*

You may be thinking or debating with your team about why we are not building the recommendation edge directly from our user to the content. Technically, that is a viable option. However, we are taking a different approach because we want to be able to provide an immediate recommendation for a user's most recent rating.

To get a conceptual understanding of how shortcut edges will be used, let's delve into how we want to use them.

## Seeing What We Designed to Deliver in Production

Thinking through what you need to be able to query in production helps you to define boundaries on the complex problem of precalculating shortcut edges. We created Figure 12-3 to illustrate what we aim to make possible in our final example.

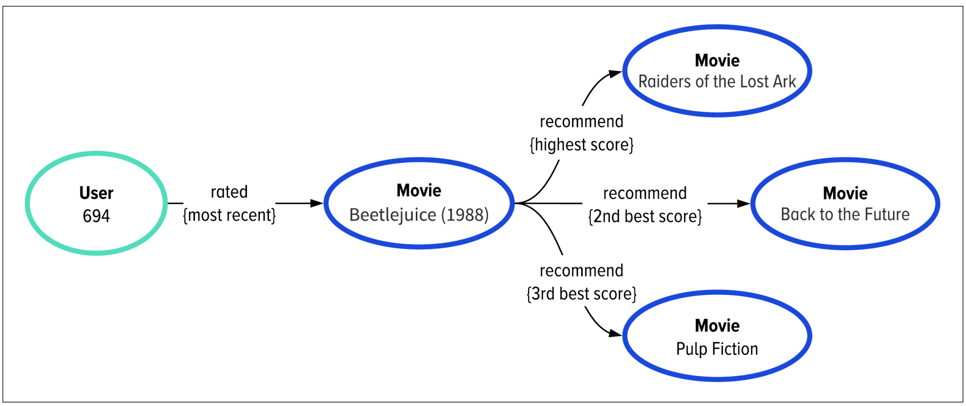



*Figure 12-3. A visual of what we are trying to precompute so that we understand what our batch job needs to calculate*

Figure 12-3 shows a conceptual model of using shortcut edges in the final query for the production version of movie recommendations. Specifically, we want to follow a user's *most recent* recommendation to generate the highest ranked set of movie recommendations. To do this, we need to precompute a shortcut edge called `recommend` that connects a user's most recent movie rating to the new content we would recommend to the user.

## Pruning: Different Ways to Precompute Shortcut Edges

The last topic we would like to address in this section is the different ways to prune and calculate shortcut edges with your data.

The main tricks when using shortcut edges come down to what and how often you precompute the aggregations they hop over. Let's talk about the techniques that we recommend you consider for your application. We will point out the decisions we made for our data along the way.

When you are first exploring how to use shortcut edges, your team will want to have a discussion on the limitations you will build into the computational process. Typically, there are three ways to limit the total amount of data that is considered for a shortcut edge: by total score, by total number of results, and by domain knowledge expectations.

Let's briefly discuss what we mean by each of these options.

#### Pruning by score thresholds

The first way you can filter out shortcut edges is with a predefined score threshold. In this approach, you would only include a shortcut edge for a recommendation if the score you calculate is above some threshold.

You already worked through the idea of using a hard threshold in a result set in this book. We walked through the use of a specific threshold when we defined the inflection point for trust in Chapter 9. We derived the specific point at which a weight above a threshold meant trust for our paths. This point is a mathematically derived limit above which a path is a trusted path for the application and below which it is distrusted.

For your recommendations and our movie data, you are not going to have such a defined fixed point.

If you want to go down this route, you will need to analyze recommendation scores for your data to understand whether a certain range of values is preferred by your user base. Regrettably, for our movie data, we do not have a specific threshold to give you when it comes to our NPS-inspired metric. But it is still important for you and your team to consider this option for your data.

In the absence of a mathematical threshold (or in combination with one), you can also limit the number of shortcut edges you include in your results.

#### Pruning with hard limits on total recommendations

The second way you can limit your use of shortcut edges is by defining the total number of edges you are going to include in production. Making the decision to store only 100 shortcut edges is an example of a hard limit. Your team can choose to include the 100 highest scoring recommendations, or you can include a selection from a range of scores.

A hard limit on the total number of edges is probably more popular than a specific score threshold for two reasons. First, it is easier to reason about the effects of the hard limit on your production application. With a hard limit, you can calculate the total amount of disk space required to store and maintain this data in production. Second, you can reasonably use them in your production query by selecting the most popular recommendation for your user. Or you can select the most highly recommended content that your user hasn't previously watched.

We are going to use the hard-limit technique for the shortcut edges we calculate for our movie data.

Once you have developed and deployed your process for shortcut edges, there is one more concept to consider to make your recommendations more relevant to your users.

#### Pruning by applying domain knowledge filters

A filter on a movie's genre to tailor recommendations to your user's preferences is an example of how to use domain knowledge to prune your recommendations.

Essentially, if your user likes dramas, you will want to also include that type of filter as you recommend new movies.

There are a myriad of topics you could explore for how domain knowledge tailors the recommendations for our movie data. The most popular ways you already experience this when you use Netflix are the recommendations by genre, specific actors, or current trends.

Ultimately, filtering your recommendations according to domain knowledge is something to plan for in your application. The application of domain knowledge filters will eventually become a component of your application as it evolves.

We will get you started with the basics first, so that you can focus on what it takes to get into production.

## Considerations for Updating Your Recommendations

When planning for delivering recommendations in production, you also need to consider how often you are going to update your shortcut edges. And your team needs to figure out how to design your pipelines to finish the computations in good time.

> Consider your own experience with recommendation windows, like the "Recommended for you" section on Netflix. How often do you log in to your account and see a refreshed list of movie recommendations? Can you tell when the section was updated due to your recent viewing history? These questions and considerations are what we mean when we mention figuring out how often to update your recommendations to provide a better user experience.

Computing a shortcut edge across all of your users' ratings is very expensive. You are going to have to reduce the scope of how often your team does these calculations. There are three tips that we are going to recommend you discuss with your team when you are designing how you build shortcut edge procedures in your application:

1. Updating the shortcut edges only for the content that has changed
2. Building data pipelines that account for successful recommendations
3. Creating robust computational processes

We are going to briefly describe what we mean for each of those tips.

First, not all content on your platform is going to be viewed or rated every day or even every week. Therefore, you do not need to recompute shortcut edges for your entire graph. Your team will want to find a way to build shortcut edges only for the most updated data to keep up with what is trending.

Second, you will want to consider what recommendations your user base actually clicks on and use this information to help identify what category of recommendations you need to recalculate. Today, you experience this within the "what's trending" section of your applications. These signals in your application are some of the most important features to capture because they represent a successful event for learning what your users like right now. With your team, plan how you are going to capture successful recommendations and use that information to account for current trends.

The last topic to consider focuses on building robust computational processes. When we say "robust," we are talking about breaking down your problem into smaller, deterministic calculations that are easily repeatable. Using smaller and more local calculations, instead of larger global calculations, allows your team to have a more agile and fault-tolerant data pipeline.

Next, let's walk through how we computed the shortcut edges for our example.

# Calculating Shortcut Edges for Our Movie Data

Shortcut edges help you get around your graph's branching factor and supernodes at query time.

What you can't get around is the amount of time that it takes to precompute shortcut edges.

For our movie data, we set up a separate environment to precompute the shortcut edges for you. This section walks you through what we did and why we made those decisions. Admittedly, there are many different ways to set up offline or batch processes to add features to your production data.

We are showing you one such approach knowing that it might not be ideal for all situations. We are going to point out different approaches and the trade-offs involved at the end of this section.

## Breaking Down the Complex Problem of Precalculating Shortcut Edges

We found that the schema and query we built up in Chapter 10 was good enough for calculating our shortcut edges.

Each query just needed more time to process all of the data.

Therefore, we broke down the process for computing shortcut edges into the following three steps:

1. Figure out the schema needed to use the NPS-inspired metric in a production graph.
2. Use the final query from Chapter 10 to create a list of shortcut edges for one movie.
3. Divide up the work to calculate shortcut edges with basic parallelism.

Let's start by taking a look at the schema we used in this environment.

### Schema required for calculating shortcut edges on our movie data

Regardless of the metric, our collaborative-filtering query simply needs movies, users, and the `rated` edge. There are two requirements for the `rated` edge. First, we need it to be sorted by the `rating` so that we can group the edges according to their rating. Second, we need to be able to traverse the edge in both directions.

These requirements give us the schema in Figure 12-4.

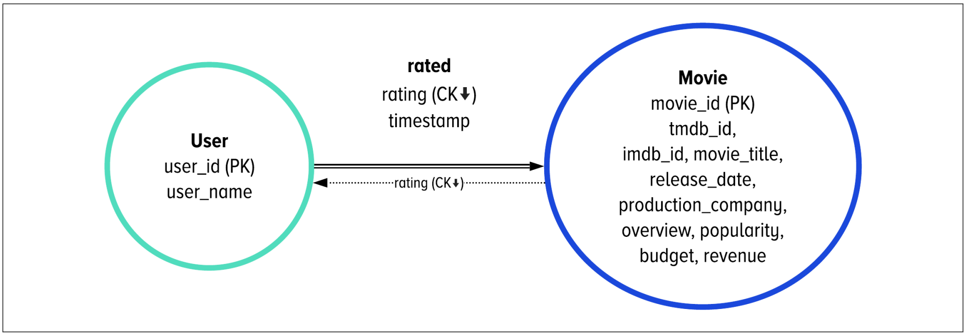



*Figure 12-4. The production schema required in the external environment for calculating shortcut edges*

The data model in Figure 12-4 describes the entire graph we constructed and loaded into the separate environment. We loaded only the movie and user vertices. We created one edge, `rated`, with a clustering key of the `rating`. Then we added a materialized view so that we could use the edge in the reverse direction in our collaborative-filtering query.

The vertex and edge labels for Figure 12-4 are as follows:

```
schema.vertexLabel("Movie").
       ifNotExists().
       partitionBy("movie_id", Bigint).
       property("tmdb_id", Text).
       property("imdb_id", Text).
       property("movie_title", Text).
       property("release_date", Text).
       property("production_company", Text).
       property("overview", Text).
       property("popularity", Double).
       property("budget", Bigint).
       property("revenue", Bigint).
       create();

schema.vertexLabel("User").
       ifNotExists().
       partitionBy("user_id", Int).
       property("user_name", Text). // Augmented, Random Data
       create();

schema.edgeLabel("rated").
      ifNotExists().
      from("User").
      to("Movie").
      clusterBy("rating", Double).
      property("timestamp", Text).
      create()
```

Figure 12-4 shows one bidirectional edge, or an edge that needs a materialized view. The code is as follows:

```
schema.edgeLabel("rated").
       from("User").
       to("Movie").
       materializedView("User__rated__Movie_by_Movie_movie_id_rating").
       ifNotExists().
       inverse().
       clusterBy("rating", Asc).
       create()
```

Next, let's use this data model to calculate our shortcut edges.

### Collaborative-filtering query to calculate shortcut edges

Given our schema, the next step is to outline how we are going to use our work building queries on the graph data from Chapter 10.

We lifted the query we developed for the NPS-inspired query and made three modifications to it:

1. We wanted to start on a movie instead of a person.
2. We wanted to limit to the 1,000 highest scoring results.
3. We needed to create a list in which each entry has the original movie, recommended movie, and NPS-inspired metric.

We used the Gremlin query that we developed in Chapter 10 with these three small adjustments. We will show you where the three modifications are after the code. Further, Example 12-1 shows the query we used to calculate 1,000 shortcut edges for a given movie. We selected 1,000 both to satisfy our upcoming queries and to provide you with an interesting set of edges to explore if you choose.

*Example 12-1.*

```
g.withSack(0.0).                                  // starting score: 0.0
  V().has("Movie","movie_id", movie_id).          // locate one movie
    aggregate("originalMovie").                   // save as "originalMovie"
  inE("rated").has("rating", P.gte(4.5)).outV().  // all users who rated it 4.5+
  outE("rated").                                  // movies rated by those users
  choose(values("rating").is(P.gte(4.5)),         // is the rating >= 4.5?
         sack(sum).by(constant(1.0)),             // if true, add 1 to the sack
         sack(minus).by(constant(1.0))).          // else, subtract 1
  inV().                                          // move to movies
  where(without("originalMovie")).                // remove the original
  group().                                        // create a group
    by().              // keys: movie vertices; will merge duplicate traversers
    by(sack().sum()).  // values: will sum the sacks from duplicate traversers
```

```
14 unfold().            // populate every entry from the map into the pipeline
15 order().             // order the whole pipeline
16   by(values, desc).  // by the values for the individual map entries
17 limit(1000).         // take the first 1000 results, which will be the top 1000
18 project("original", "recommendation", "score"). // structure your results
19   by(select("originalMovie")).                  // "original": original movie
20   by(select(keys)).                             // "recommendation": rec movie
21   by(select(values)).                           // "score": sum of NPS metrics
22 toList()                                        // wrap the results in a list
```

Let's point out the three places in Example 12-1 that are changes from the query we developed in Chapter 10. First, line 2 in Example 12-1 shows how we started at a specific movie. Then we followed the same process for performing collaborative filtering and calculating an NPS-inspired metrics from line 3 to line 16.

The last two changes in Example 12-1 are about formatting the results for later use. Line 17 shows our second change: we reduced the total number of results to include only the 1,000 highest scoring recommendations. This limitation is vital because this type of approach will eventually compute an edge for all 327,000+ other movies in the database as you collect enough ratings. Then lines 18 through 22 in Example 12-1 show how we formatted our results to make it easy to save our work into our production environment. We created a list that had 1,000 entries with this structure: original movie, recommended movie, and then the NPS.

To give you an idea of what the results looked like, Figure 12-5 shows the top six recommendations for one of our movies.

| index ↑ | Original | Recommendation | Score |
|---|---|---|---|
| 0 | Aladdin (1992) | Lion King The (1994) | 4911.0 |
| 1 | Aladdin (1992) | Shawshank Redemption The (1994) | 4697.0 |
| 2 | Aladdin (1992) | Beauty and the Beast (1991) | 4624.0 |
| 3 | Aladdin (1992) | Forrest Gump (1994) | 4310.0 |
| 4 | Aladdin (1992) | Toy Story (1995) | 4186.0 |

*Figure 12-5. The top six shortcut edges calculated for movie `588` during our batch process*

The original movie in Figure 12-5 is movie `588`, *Aladdin*. The top five recommendations that we computed and saved as shortcut edges included *The Lion King*, *The Shawshank Redemption*, *Beauty and the Beast*, *Forrest Gump*, and *Toy Story*.

Now that you see the schema and query that we used, let's talk about how we divided up the work to get it done.

### Using simple parallelism to divide up the work

We opted to decompose the larger process of precalculating shortcut edges for our entire graph into smaller, independent problems. We can break down movie recommendations into many smaller queries because each movie's set of recommendations is independent from any other movie's set.

Example 12-2 outlines how we divided up computing shortcut edges for our data into many smaller, independent queries.

*Example 12-2.*

```
SETUP: Load the users, movies, and ratings graph into a separate environment
DECOMPOSE: Divide the movie into _ids into N smaller, independent lists
ASSIGNMENT: Assign one list per processor
ORCHESTRATION: Synchronously compute shortcut edges for each movie
EXTRACTION: Save the results to be loaded into the production graph
```

The approach described in Example 12-2 employs a straightforward and basic way to divide up the work required to calculate shortcut edges for our movies. We started by setting up a separate environment and loading just the users, movies, and ratings into that environment. Then we divided a list of `movie_ids` into N separate and independent groups. We assigned each list to a separate process so that we could use basic parallelism to compute the shortcut edges for an individual movie, synchronously. Last, we saved the results into a list that we could then load into our production model.

> Breaking up the calculation of shortcut edges into many smaller queries follows a process known as data parallelism. You can use data parallelism when the same computation needs to be performed on different subsets of the same data. Essentially, you use the same model for each thread in your computing environment, but the data given to each of them is divided and shared. We recommend Vipin Kumar's book on the topic if you want to learn more.[1]

The approach we outlined in Example 12-2 prioritizes the minimization of computation time over memory. We can break down movie recommendations into many smaller queries because each movie's set of recommendations is independent. Some complex problems, like PageRank, cannot be decomposed in this same manner.

1 Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to Parallel Computing*. 2nd ed. (Boston: Addison-Wesley Professional, 2003). *https://www.oreilly.com/library/view/introduction-to-parallel/0201648652/*.

Before we can show you how to use this in your production graph, we need to have a brief side discussion about another very common way to solve this same problem.

## Addressing the Elephant in the Room: Batch Computation

We had to make some trade-offs when deciding how we were going to approach the computation of shortcut edges for this book. As you just learned, we decided to use basic parallelism to divide up the work and computed shortcut edges for each individual movie independently.

However, the Gremlin query language also has a batch execution model that is primarily used for larger batch computations across large portions of the graph.

So why didn't we use batch computation to precompute the shortcut edges?

For this book, the reason is primarily one of scope. Using batch computation introduces enough depth and complexity to fill another book. So we are just providing a teaser here for batch graph queries and will leave you with the exciting realization that there is more to learn about applying graph thinking than we were able to cover in this book.

If you are deciding between parallelized transactional queries and batch computation for your shortcut edge computation, here are some of the trade-offs that you should consider.

#### Examples of when batch computation may be better for your environment

Gremlin queries that execute batch computations can exploit shared computations. This can result in quicker overall execution because of not having to traverse the same parts of the graph twice. For instance, in our example we have a lot of movies that were rated by the same reviewer. With transactional queries, we traverse through those reviewer vertices multiple times. With batch computation, we can bulk those computations together.

Batch computation usually requires more resources (in particular memory), which can interfere with a concurrent transactional workload. For instance, in our example a concurrent batch query may put enough stress on the database to delay concurrently running recommendation retrieval queries. This pressure could lead to longer latencies and a worse user experience. For that reason, batch computations are often started either when there is little load on the database or in a separate data center and Cassandra cluster, but that may not be an option for you.

With DataStax Graph, batch computations can be done in an analytical data center (of the same cluster). Then, once precomputed results are written back into the graph, they are also automatically replicated to the operational data center. This is how workload separation works with Apache Cassandra and DataStax Graph.

#### Examples of when transactional queries may be better for your environment

With batch computation, you are always recomputing all shortcut edges, which gives that approach the computational advantage. But in many cases, you may want to update some shortcut edges more frequently than others and need the flexibility that the transactional approach provides.

Transactional queries allow for more selective updates of precomputed edges.

For instance, in our example the shortcut edges for recent movies are becoming stale more quickly as new ratings start pouring in. In that scenario, we would want to recompute those edges more frequently than we would for old movies, where there is little to no change. For a second scenario, maybe your precalculation job fails and you have to start over. Using smaller transactional queries would be easier to track and restart than having to recompute the whole graph.

The transactional approach of data parallelism works better if there are fewer starting points. In our example, we have thousands of movies to start from, which is a rather small number. If that number were in the millions, then the transactional approach would take a long time (and be pretty error-prone), which would favor the batch approach.

There are other trade-offs that depend on your particular situation, environment, and infrastructure, but those are the main ones to consider in making this decision.

We chose the data parallelism approach with transactional queries to show you how we reasoned about calculating shortcut edges for this example. It isn't necessarily the best approach for all situations. You will need to consider your environment and your application's expectations when determining how to set up precomputing shortcut edges for your next project.

Now that you have an idea of how we computed the shortcut edges, let's show the recommendation data model that we used in production, load the data, and walk through our queries one last time.

## Production Schema and Data Loading for Movie Recommendations

Recall a few pages back when we showed you our conceptual design for how we wanted to deliver our recommendations. Figure 12-3 showed how we wanted to query a user's most recent rating and then use the top recommendations to provide new content to our user. The details we walked through in that image give us the outline for how we want to deliver recommendations in our environment.

Let's walk through the schema we will use and the final data loading processes for this last example.

## Production Schema for Movie Recommendations

One of the most important details of our conceptual visualization in Figure 12-3 was shown underneath the edges. We saw that we wanted to use a user's most recent rating to deliver the three highest scoring recommendations. These constraints describe how we can cluster our edges most optimally for performance.

Figure 12-6 is the final schema model that we will use for our recommendations. We will use the same two vertex labels as we had for our shortcut edges: users and movies.

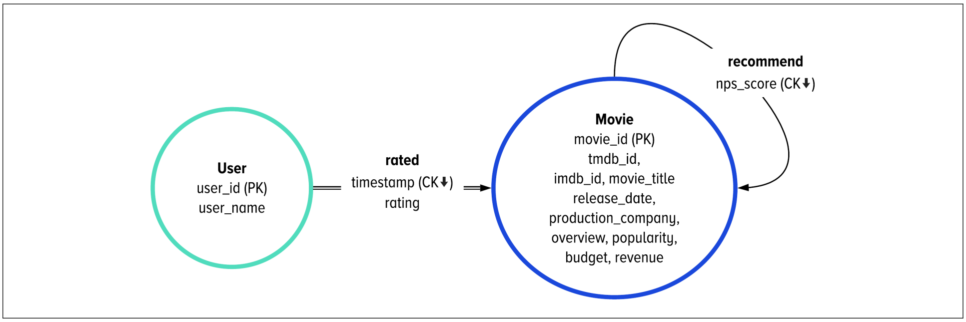



*Figure 12-6. The conceptual model of the schema we will use for the production version of our recommendation system*

The differences in the two models in this chapter are the edge labels. The user's ratings will be clustered by time so that we have easy access to the most recent rating. Then we will use the shortcut edges we precomputed in "Calculating Shortcut Edges for Our Movie Data" on page 357 to directly recommend movies from a given movie. The shortcut edges will be stored as the `recommend` edge, sorted by their rating.

Example 12-3 shows the vertex labels, and Example 12-4 shows the edge labels.

*Example 12-3.*

```
schema.vertexLabel("Movie").
       ifNotExists().
       partitionBy("movie_id", Bigint).
       property("tmdb_id", Text).
       property("imdb_id", Text).
       property("movie_title", Text).
       property("release_date", Text).
       property("production_company", Text).
       property("overview", Text).
       property("popularity", Double).
       property("budget", Bigint).
       property("revenue", Bigint).
       create();
```

```
schema.vertexLabel("User").
       ifNotExists().
       partitionBy("user_id", Int).
       property("user_name", Text). // Augmented, Random Data
       create();
```

*Example 12-4.*

```
schema.edgeLabel("rated").
      ifNotExists().
      from("User").
      to("Movie").
      clusterBy("timestamp", Text, Desc).  // Note: changed clustering key
      property("rating", Text).
      create()

schema.edgeLabel("recommend").
      ifNotExists().
      from("Movie").
      to("Movie").
      clusterBy("nps_score", Double, Desc).
      create()
```

The last part of our setup is to walk through how to load the data.

## Production Data Loading for Movie Recommendations

The user and movie vertices will be loaded the same way as we walked through in Chapter 10. We are going to skip that part of the loading process because it is exactly the same, with the same files.

The only new data to load is the shortcut edges for the `recommend` edge label. We created a `csv` file of all of the precomputed edges so that we can load them easily into our graph for our final production recommendation queries.

We created a file to load with all of the precomputed shortcut edges in this chapter. The file structure is shown in Table 12-1.

*Table 12-1. Six shortcut edges found in the accompanying csv file for this example*

| out_movie_id | in_movie_id | nps_score |
|---|---|---|
| 588 | 364 | 4911.0 |
| 588 | 318 | 4697.0 |
| 588 | 595 | 4624.0 |
| 588 | 356 | 4310.0 |
| 588 | 1 | 4186.0 |
| 588 | 593 | 3734.0 |

As you have seen many times in this book, the hardest part of structuring your edge files is making sure your data, header, and graph schema all line up. The header line of Table 12-1 shows that the first `movie_id` on each line corresponds to the `out_movie_id`. The `out_movie_id` is the movie for which we computed a recommendation. The second `movie_id` on each line corresponds to the `in_movie_id`. This second identifier connects the edge to the recommended movie. The last piece of data on each line is the `nps_score` for the recommendation that we already computed for you using the NPS-inspired collaborative-filtering approach.

If you want to confirm that the header, data, and schema all align, you can inspect the schema of the `recommend` edge label's table definition.

The final step is to load the shortcut edges into your graph. Example 12-5 shows the command that loads the data using the bulk loading tool.

*Example 12-5.*

```
dsbulk load -g movies_prod
            -e recommend
            -from Movie
            -to Movie
            -url "short_cut_edges.csv"
            -header true
```

Let's walk through the final version of our recommendation queries to see how we will use these shortcut edges to deliver recommendations in a few steps.

## Recommendation Queries with Shortcut Edges

We designed our schema and precomputed our shortcut edges so that we could deliver our recommendations to our end user as quickly as possible. Going through all of the work ahead of time ensures that your application delivers the fastest and best user experience.

In this last section, we want to do three things. First, we want to check that our loaded shortcut edges match what we computed during our offline process. The second section shows you how to use these shortcut edges in three different styles of recommendation queries. The last section shows you how to reason about query performance by mapping the number of edge partitions accessed during two of our three production queries.

Let's first confirm that our shortcut edges match the computation we showed you in "Calculating Shortcut Edges for Our Movie Data" on page 357.

## Confirming Our Edges Loaded Correctly

We took a snapshot of the shortcut edges we computed for the movie *Aladdin* in "Calculating Shortcut Edges for Our Movie Data" on page 357. We know that *Aladdin*'s `movie_id` is 588, so in Example 12-6, let's query for *Aladdin*'s top five recommendations to make sure they match our expectations.

*Example 12-6.*

```
1 g.V().has("Movie", "movie_id", 588).as("original_movie").
2      outE("recommend").
3      limit(5).
4      project("Original","Recommendation", "Score").
5        by(select("original_movie").values("movie_title")).
6        by(inV().values("movie_title")).
7        by(values("nps_score"))
```

Example 12-6 applies the Gremlin patterns we have used throughout this book. Line 1 starts with a partition key lookup to *Aladdin*'s movie vertex. Then we walk to the `recommend` edges on line 2.

Line 3 of Example 12-6 brings together two very important concepts: clustering keys in Apache Cassandra and limits in Gremlin. Recall that the `recommend` edges are clustered by their rating. Therefore, the use of `limit(5)` on the edge table finds the top five recommendations according to their rating score, because it selects the first five rows of the partition in the underlying tables. This is exactly why Gremlin with distributed adjacency lists in Apache Cassandra is so fast.

The remaining work in lines 4 through 7 of Example 12-6 nicely formats the results. We create a user-friendly structure that lists *Aladdin*, the title of the recommended movie, and its score. Figure 12-7 shows you what you will see in the accompanying Studio Notebook.

| index ↑ | Original | Recommendation | Score |
|---|---|---|---|
| 0 | Aladdin (1992) | Lion King The (1994) | 4911.0 |
| 1 | Aladdin (1992) | Shawshank Redemption The (1994) | 4697.0 |
| 2 | Aladdin (1992) | Beauty and the Beast (1991) | 4624.0 |
| 3 | Aladdin (1992) | Forrest Gump (1994) | 4310.0 |
| 4 | Aladdin (1992) | Toy Story (1995) | 4186.0 |

*Figure 12-7. Confirming that we properly loaded our shortcut edges from our first query in Example 12-6*

Figure 12-7 matches the expected top five recommendations for the *Aladdin* movie that we previewed in "Calculating Shortcut Edges for Our Movie Data" on page 357.

Let's now use these edges to show you how to deliver recommendations to a specific user.

## Production Recommendations for Our User

There are three queries we want to show you in this section. They are:

1. Query 1: The top three recommendations for the most recent rating by our user
2. Query 2: The top recommendation for the three most recent ratings by our user
3. Query 3: Combining 1 and 2 to deliver the top three recommendations for each of the three most recent ratings by our user

The first two queries take different approaches to providing three recommendations to the end user. The last approach was designed to show you how to get nine specific recommendations by using barrier steps in Gremlin.

### Query 1: The top three recommendations for the most recent rating by our user

We designed the schema, processes, and shortcut edges with one purpose in mind: to be able to immediately deliver new content according to a user's most recent rating. We can do that by accessing our user's most recently rated movie and then accessing our top three precomputed recommendations. Example 12-7 shows how to do this in Gremlin.

*Example 12-7.*

```
1 g.V().has("User","user_id", 694).      // our user
2       outE("rated").limit(1).inV().    // first "rated" edge is most recent
3       outE("recommend").limit(3).      // first three are top 3 recommendations
4       project("Recommendation", "Score").  // create a map with two keys
5         by(inV().values("movie_title")).   // move to the movies; get title
6         by(values("nps_score"))            // stay on edges; get score
```

The Gremlin steps in Example 12-7 are all ones that we have used before. The beauty of this example is shown on lines 2 and 3 with the use of `limit(x)` on the edges.

On line 2, recall that the `rated` edges are clustered by time. Therefore, `outE("rated").limit(1)` accesses the first edge in the partition, which is also the most recent rating.

This same access pattern is used on line 3 with `outE("recommend").limit(3)` because the `recommend` edges are sorted on disk by rating. Lines 4 through 6 use the

project step to create user-friendly formats of the data. The results of this query are shown in Figure 12-8.

| index | Recommendation | Score |
|---|---|---|
| 0 | Rear Window (1954) | 85.0 |
| 1 | Casablanca (1942) | 78.0 |
| 2 | Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) | 77.0 |

*Figure 12-8. Seeing the three new content recommendations according to our most recent rating in Example 12-7*

Figure 12-8 shows that the three newest movies to recommend to our user, according to their most recent movie rating, are *Rear Window*, *Casablanca*, and *Dr. Strangelove*. Interesting choices, user `694`.

Seeing that the scores for these movies are relatively low, you may want to broaden the set of movies to recommend by considering more ratings. Let's take a look at how to diversify your ratings with Query 2.

### Query 2: The top recommendation for the three most recent ratings by our user

The goal of this next example is still to provide three recommendations to our user, but to find them a bit differently. We want to query our user's three most recent ratings and provide the top recommendation for each of them. Example 12-8 shows how we will do this in Gremlin.

*Example 12-8.*

```
1 g.V().has("User","user_id", 694).   // our user
2       outE("rated").limit(3).inV(). // three most recently rated movies
3       project("rated_movie", "recommended_movie", "nps_score"). // map w/ 3 keys
4         by("movie_title").          // value for the key "rated_movie"
5         by(outE("recommend").       // value for the key "recommended_movie"
6            limit(1).                // first recommendation is top rated
7            inV().values("movie_title")). // traverse to the movie and get title
8         by(outE("recommend").       // value for the key "nps_score"
9            limit(1).                // first recommendation is top rated
10           values("nps_score"))     // stay on the edge; get the score
```

The beauty of Example 12-8 is that it shows you how to create more diversity in your recommendation set. On line 2, we use the fact that the `rated` edges are clustered by time, but this time we collect the three most recent ratings with `limit(3)`. Then, for each of the three most recent ratings, we want to find the top recommended movie.

This is what we do on lines 5 and 6. We limit each traverser to its top recommendation and then access the movie title. The rest of Example 12-8 formats different pieces of the query so that we collect a meaningful view of the result data. Figure 12-9 shows our results.

| index ↑ | rated_movie | recommended_movie | nps_score |
|---|---|---|---|
| 0 | Safety Last! (1923) | Rear Window (1954) | 85.0 |
| 1 | Bill & Ted's Excellent Adventure (1989) | Back to the Future (1985) | 691.0 |
| 2 | Overboard (1987) | Shawshank Redemption The (1994) | 52.0 |

*Figure 12-9. Seeing a different way to find three new content recommendations according to our user's most recent ratings*

Figure 12-9 shows one of the recommendations from our first query, `Rear Window`. We now see that this movie is the top recommendation for a movie titled `Safety Last!` The other two movies that user `694` has recently rated are also shown in Figure 12-9 along with their top recommendation.

By broadening to a larger set of recent ratings for our user, we were able to find a fairly popular movie: *Back to the Future*. This movie has the highest NPS-inspired metric and therefore is also the most popular movie in our set of recommendations.

#### Query 3: The top three recommendations for each of the three most recent ratings by our user

One last natural question to ask about this data would be to collect, say, the top three recommendations for each of our user's three most recent ratings. For each rating, we will want to move to the set of recommended movies and ask each traverser to find three. We do this in Gremlin with the `local()` scope around the traversers. Then we want to bring all recommendations back together and merge them into one list. Let's make that our last example for this dataset in Example 12-9.

*Example 12-9.*

```
g.V().has("User","user_id", 694).        // our user
     outE("rated").limit(3).inV().       // 3 most recent rated movies
     local(outE("recommend").limit(3)). // top 3 recommendations for each movie
     group().                            // create a map
       by(inV().values("movie_title")). // keys for the map; merge duplicates
       by(values("nps_score").sum()).   // values; sum values for duplicates
     order(local).                       // sort the map
       by(values, desc)                  // by its values, descending
```

Example 12-9 starts off the same as Example 12-8 but introduces the use of local scope on line 3. Using local scope ensures that each traverser grabs three recommendations to populate into the map we construct on line 4. Line 5 shows us that the keys of this map will be the movie titles. Line 6 aggregates all of their ratings into one value. Figure 12-10 shows the results.

| index ↑ | keys | values |
|---|---|---|
| 0 | Back to the Future (1985) | 737.0 |
| 1 | Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981) | 665.0 |
| 2 | Matrix The (1999) | 660.0 |
| 3 | Rear Window (1954) | 85.0 |
| 4 | Casablanca (1942) | 78.0 |
| 5 | Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) | 77.0 |
| 6 | Shawshank Redemption The (1994) | 52.0 |
| 7 | Forrest Gump (1994) | 42.0 |

*Figure 12-10. The top three recommendations for each of the three most recent ratings by our user*

Figure 12-10 shows the movies to recommend to our user along with each movie's final score. It is interesting to note that we do not have nine total recommendations. Figure 12-10 has eight results because *Raiders of the Lost Ark* showed up as a recommendation for the movie *Overboard* and the movie *Bill and Ted's Excellent Adventure*; the final score for *Raiders of the Lost Ark* aggregated the score from each recommendation.

We encourage you to play around with these queries in the accompanying notebook. Most notably, take a look at what happens when you query this data with and without `fold()`. How would you expect the structure of the results to change when you remove the barrier? Did you get it right?

#### What are we ignoring that you need to consider?

There are three additional topics you will want to consider for your application.

First, the most obvious filter you will want to include in your application would limit recommendations according to a user's preference. Such a filter could remove movies they have already watched or have rated poorly.

The second topic to consider is the size of the result set. We arbitrarily chose three recommendations, but we precomputed 1,000 recommendations per movie. We used the smaller number in our examples to illustrate the concepts in a meaningful way. You can explore sampling the 1,000 edges to compare different ways to use them.

The most popular way to expand beyond the top three recommendations is also our last tip for your recommendation query. After your user views a small number of recommendations, you will want your application to keep scrolling and pull more data. You will want to set up your application to be able to stream more results to your end user, which is why you will likely need to go deeper into your set of shortcut edges.

Hopefully, throughout all of the exercises across this book, you have learned how to apply limits and filters to accommodate any of these options to the type of recommendations you deliver to your users.

## Understanding Response Time in Production by Counting Edge Partitions

The synthesis of `limit(x)` in Gremlin, combined with your graph schema's distributed architecture, is one of the most important concepts in this book. This last section emphasizes that point to really bring it home.

The queries we introduced in the preceding section each provided a different set of recommendations to your end user. We saw that the results of Query 2 and Query 3 were more diverse than the results of Query 1.

Thus you may have concluded that queries like Query 2 or Query 3 are better for your application because they provide more selection for your user. And they may be.

But you have one last concept to synthesize in order to understand the performance trade-offs of Query 1, Query 2, and Query 3.

The performance implications for each of our queries come down to how many edge partitions are accessed to deliver the recommendations. And one of our queries has a significant advantage when it comes to performance.

Let's synthesize the traversals we have shown here with the concepts we detailed in Chapter 5 by laying out the number of edge partitions required in each traversal. We start by showing the number of edge partitions required for our first query.

#### Partitions traversed for Query 1: The top three recommendations for the most recent rating by a user.

Figure 12-11 shows that we need to access two separate edge partitions to deliver our three recommendations.

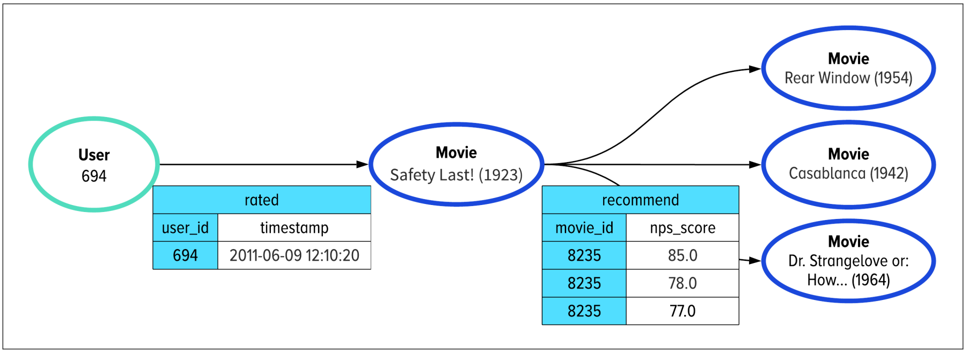



*Figure 12-11. Understanding the number of edge partitions accessed by Example 12-7; the query uses two different edge partitions*

The first edge partition is from user `694` to the movie vertex for `Safety Last!` The second edge partition is from the single movie vertex to its three top-rated movies. The tables drawn alongside the graph data in Figure 12-11 highlight exactly when different edge partitions are accessed during our traversal.

#### Partitions traversed for Query 2: The top recommendation for the three most recent ratings by a user.

Let's contrast the number of partitions required for Query 1 by looking at the number of edge partitions required for Query 2 in Figure 12-12.

Figure 12-12 shows that we need to access four separate edge partitions to deliver our three recommendations. The first partition is the same as we used before: the user's ratings edges. However, this time we selected three separate movie vertices. To access the top recommendation for each movie, we have to look at three different partitions. Therefore, Query 2 requires four different edge partitions to find three different recommendations.

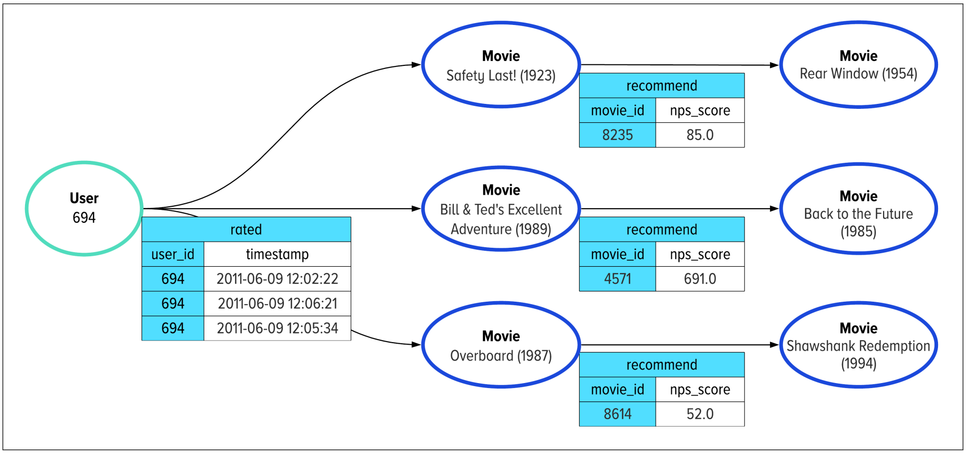



*Figure 12-12. Understanding the number of edge partitions accessed by Example 12-8; the query uses four different edge partitions*

#### Partitions traversed for Query 3: The top three recommendations for each of the three most recent ratings by our user.

The last query to consider is Query 3, and like Figure 12-12, Figure 12-13 shows that we need to access four separate edge partitions in order to deliver our recommendations.

The first partition is the same as we used before: the user's ratings edges. However, this time, we selected three separate movie vertices. To access the top three different recommendations for each of our movies, we have to look at three different partitions. Therefore, Query 3 requires four different edge partitions to find three different recommendations.

You also see in Figure 12-13 that the second and third movies both recommend `Back to the Future`. Look back to our results listed in Figure 12-10. We can see here that the score for `Back to the Future` aggregated both `nps_scores`: `691.0 + 52.0 = 737.0`.

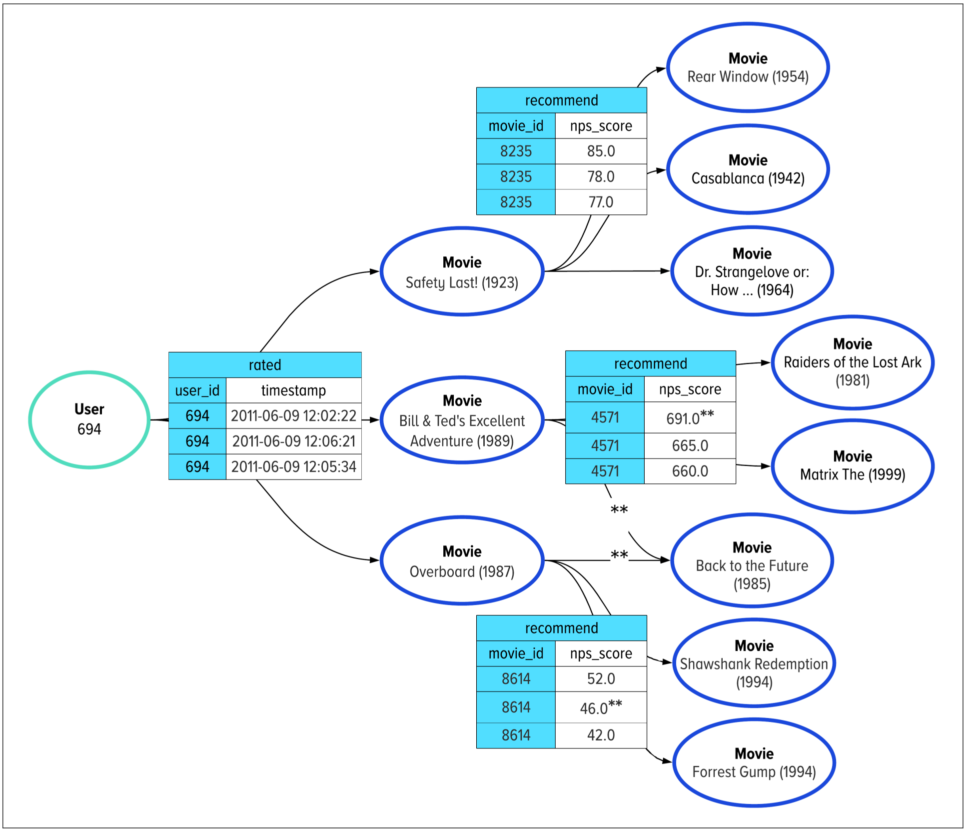



*Figure 12-13. Understanding the number of edge partitions accessed by Example 12-9; the query uses four different edge partitions*

## Final Thoughts on Reasoning About Distributed Graph Query Performance

The key for understanding performance of a query in a distributed environment lies in synthesizing two concepts. As we just walked through, there is a direct correlation to a query's speed according to the number of partitions it requires across a distributed environment.

With practice, it will become easier to follow the number of partitions accessed in your query. Continue thinking about and visualizing your queries like what we illustrated in Figure 12-13 to build up your understanding.

The second main contributing factor to your query's performance is your data's connectivity. We used shortcut edges in this chapter to simultaneously mitigate your data's branching factor and potential supernodes.

We have constructed all the information in the past few sections so that you can reason about your query's performance. Altogether, your graph query's performance is an intricate balance between distributed partition management and planning for your data's branching factor. At the end of the day, these are all the foundational concepts you need to practice in order to be able to reason about the performance of distributed graph queries. We hope you found them instructive.

# CHAPTER 13
# Epilogue

We are incredibly honored that you went with us on this journey to graph thinking and its application to complex problems. You learned a new way of thinking for solving complex problems, a new body of theory to formalize that thinking, and a number of new techniques and technologies for applying that thinking in building practical solutions.

As Leonardo da Vinci said, "A developer would be overcome by sleep and hunger before being able to describe with words what a code sample can express in an instant."

Like with all crafts, mastery of graph thinking can be gained through continued practice. We set up our notebooks and example problems to show you how to get started with your new craft. Feel free to keep playing with those notebooks and adjusting them to suit your particular problems.

We would like to encourage you to apply the frameworks from this book to the problems that you encounter. The first chapters of this book showed you how to reason about which problems benefit from graph thinking. The criteria we walked through aren't hard and fast rules but simply rough guidelines for discerning when a problem has the characteristics that make it suitable for graph thinking. Over time, you will build an intuition that will support this decision making.

As you are starting out, it will likely feel more natural and comfortable to think about your data problems through the relational lens of tabular data. Push through the discomfort of adopting a different perspective and give graph thinking a try, especially when the relationships and connective structure of the data are important to the problem at hand.

There is nothing wrong with the relational perspective to representing data, and we are not trying to argue that graph thinking is better. It's different, and for a certain

class of problems, it's easier and more effective for finding a solution. Mastering both perspectives is critical to solving complex problems since they often need to be broken down into subproblems that require a combination of both perspectives.

As you are starting to apply graph thinking to your problems, we encourage you to follow the "development first, production second" approach we took throughout the latter chapters of this book. In other words, start with exploring your data as a graph and quickly iterate toward applying and refining suitable graph techniques before you delve deep into fine-tuning those techniques for production use. Chapter 4 through Chapter 12 walked you through how we approach this mentality for applying graph thinking to the most common connected data problems: exploring neighborhoods, branching in trees, finding paths, collaborative filtering, and entity resolution.

Think of those techniques as Lego bricks that you can combine and assemble in various ways to build the solution that works for your particular application.

# Where to Go from Here?

Is this all there is to know about graph thinking?

Far from it—this is just the end of the beginning.

Graph thinking is an incredibly rich topic, with relationships to many other areas in computer science, physics, mathematics, biology, and beyond. Once you become comfortable with viewing a problem through the lens of vertices connected by edges, you'll be surprised by the depth of understanding that this change in perspective unlocks in various areas of human inquiry.

We recommend four avenues you can take to continue your journey with graph thinking: graph algorithms, distributed graphs, graph theory, and network theory. This list is by no means comprehensive, but just a rough outline of the many learning roads you can take from here.

We are going to end with a brief section on each of these four topics and our recommendations on what to read next.

## Graph Algorithms

There is another class of graph problems to mention: graph algorithms. Unlike the specific production traversals taught in this book, graph algorithms typically require analyzing the entire graph's structure, like computing a specific analytic about the connectedness of your data.

Collaborative filtering, which we first saw in Chapter 10, is one example of a graph algorithm. Other popular graph algorithms are all-pairs shortest path, PageRank,

graph coloring, connected components identification, betweenness centrality, graph partitioning, and modularity.

There are two main concepts to mention about graph algorithms.

The first point is that a graph algorithm typically requires global computation across most of the graph, if not the entire graph. We teased the introduction of batch computation as an alternative way to use the Gremlin query languge for global computations on graph-structured data.

The second point is to acknowledge that some graph algorithms can be broken down into many localized computations, whereas others cannot. We saw one graph algorithm, namely collaborative filtering, that can be solved either with a global distributed computation or with many localized computations.

Most of the more popular global graph algorithms, like PageRank and Connected Components, cannot be decomposed into smaller computations and require distributed batch computation when applied to very large graphs. For this class of graph algorithms, it may be necessary to run the graph computation in a batch computing (or bulk synchronous) mode that distributes the computation across multiple machines in a cluster.

We recommend two books if you are interested in learning more about global graph algorithms. First, we recommend studying *Distributed Graph Algorithms for Computer Networks* by K. Erciyes (Springer) f you want to dive deeply into how and when graph algorithms can be broken down into smaller localized problems. Second, the hands-on practitioner may appreciate the code examples for the most popular algorithms in *Graph Algorithms: Practical Examples in Apache Spark and Neo4j* by Mark Needham and Amy E. Hodler (O'Reilly).

## Distributed Graphs

This book places an emphasis on distributed graphs. Graphs need to be distributed when they are too large to reasonably fit on a single machine, because of workload requirements (i.e., achieving a certain throughput at low latency), or to account for geo-distribution requirements of the data. Distributed graphs are particularly challenging because you are combining the complexity of distributed data with the complexity of graph thinking.

While DataStax Graph in Cassandra handles a lot of this complexity on behalf of the user, such as data replication and fault tolerance, understanding how distributed systems work in detail is critical to understanding the behavior of the systems under extreme conditions.

Some elements that we have not addressed in any detail in this book have to do with data consistency. DataStax Graph uses an eventual consistency model that favors sys-

tem uptime over strong consistency guarantees. Other graph databases make the opposite choice and provide stronger consistency guarantees with a higher likelihood of database unavailability.

What is right for your application depends on your business requirements. In any case, it is important to understand what consistency and availability guarantees are being provided by the system and how to reason about them.

Distributed databases are fascinating systems, and their discussion fills entire books. We encourage our readers to learn more about them. To learn more about Cassandra, the distributed database underlying DataStax Graph, we recommend *Cassandra: The Definitive Guide* by Jeff Carpenter and Eben Hewitt (O'Reilly). For a more general discussion of distributed databases, we recommend the *Principles of Distributed Database Systems* textbook by M. Tamer Özsu and Patrick Valduriez (Springer Science +Business Media).

## Graph Theory

There is a whole branch of mathematics called graph theory, which is dedicated to the study of graph structures; many of the terms introduced in this book stem from graph theory. From a practitioner's standpoint, it is most useful to familiarize yourself with the terminology and develop a basic understanding of the distinctions that graph theory draws.

If you'd like to get a deeper understanding of the terminology and basic concepts underlying graph thinking, we encourage you to study graph theory. Graph theory will teach you about certain classes of graphs, such as planar graphs, and what characteristics these graphs have. You'll learn more about the famous "graph coloring" problem.

A good starting point for your self-directed tour of graph theory is *Introduction to Graph Theory* by Richard J. Trudeau (Dover). Also, you can find a lot of introductory material on graph theory online, including an entire YouTube channel by Sarada Herke dedicated to graph theory and discrete mathematics.

When searching for content online about graph theory, you will quickly run into the term *network theory*.

## Network Theory

*Network* is a term used synonymously with *graph*, and network theory is the application of graph theory to the real world. Network theory studies natural graphs, or graph structures that occur in the real world around us and within different disciplines.

For example, sociologists apply network theory to study social networks and reason about natural connected structures. Biologists look at graphs that occur in the biological world, such as food networks (or "who eats whom?"), and within human beings, such as molecular pathways or protein-protein interaction networks.

One fascinating finding from network theory is that a lot of naturally occurring networks are "scale-free," and the degree distribution of the vertices on those networks has a power law distribution. Simply put, there are a few vertices in the graph that have a whole lot of edges, and then very many vertices with only a few edges. Twitter is a good example of a scale-free graph: there are very few people on Twitter with millions of followers, and millions of people with only a few followers.

A wide variety of natural networks are scale-free; this is the reason that supernodes exist in graphs.

One popular theory trying to explain the prevalence of scale-free networks, called the *preferential attachment* theory, speculates that as new vertices join a network over time, they are more likely to build edges to vertices that already have a lot of edges. In other words, it's a classic "the rich get richer" phenomenon.

This intuitively holds true for Twitter: if a new user joins Twitter, they are more likely to follow somebody popular like Barack Obama than a random person.

Network theory has a lot to say about natural graphs and the dynamics that shape them. This is relevant for graph practitioners to ensure that the systems we build work well on the graphs we are targeting. We already saw how important it is to be aware of and work around supernodes. Network theory helps us understand when and how supernodes arise. Similarly, there are many other topics within network theory that can give us a better understanding of certain domain graphs.

*Linked: The New Science of Networks* by Albert-László Barabási (Perseus), the scientific father of the preferential attachment theory, is a good popular science introduction to graph thinking. If you are not afraid of a denser read or of getting into the mathematics of it all, we recommend the survey paper "The Structure and Function of Complex Networks" by Mark Newman.[1] This paper provides a high-level introduction to many areas within network science with enough mathematical depth to become practical, while remaining high-level enough to cover a lot of ground quickly. And it contains a lot of references to more in-depth materials.

1 SIAM REview 45, no. 2 (2003): 167-256.

## Stay in Touch

If you've enjoyed this introduction to practical graph thinking and want to learn more about it or join a group of like-minded individuals on a shared journey, we encourage you to:

1. Follow us on Twitter: @Graph_Thinking
2. Visit our GitHub: *https://github.com/datastax/graph-book*

# Index

## C

## D

## E

## F



## G

## H



## I



## J



## K

## L



## M

## Q

## R

## S

## T

## U



## V



## W

## About the Authors

As Chief Data Officer of DataStax, **Dr. Denise Koessler Gosnell** applies the thought processes in this book to make more informed decisions with data. Prior to this role, Dr. Gosnell joined DataStax to create and lead the Global Graph Practice, a team that builds some of the largest distributed graph applications in the world. Dr. Gosnell earned her Ph.D. in computer science from the University of Tennessee as an NSF fellow. Her research coined the concept social fingerprinting by applying graph algorithms to predict user identity from social media interactions.

Like this book, Dr. Gosnell's career centers on her passion for examining, applying, and advocating the applications of graph data. She has patented, built, published, and spoken on dozens of topics related to graph theory, graph algorithms, graph databases, and applications of graph data across all industry verticals. Prior to her role with DataStax, Dr. Gosnell worked in the healthcare industry, where she contributed to software solutions for permissioned blockchains, machine learning applications of graph analytics, and data science.

**Dr. Matthias Broecheler** is the Chief Technologist of DataStax and an entrepreneur with substantial research and development experience. Dr. Broecheler's work focuses on disruptive software technologies and understanding complex systems. He is known as an industry expert in graph databases, relational machine learning, and big data analysis in general. He is a practitioner of lean methodologies and experimentation to drive continuous improvement. Dr. Broecheler is the inventor of the Titan graph database and a founder of Aurelius.


## Colophon

The animal on the cover of *The Practitioner's Guide to Graph Data* is the Mediterranean rainbow wrasse (*Coris julis*). This colorful fish inhabits the Northeastern Atlantic from Sweden to Senegal and into the Mediterranean. It lives near the shoreline and favors rocky, grassy areas. It feeds on small crustaceans such as shrimp and sea urchins and gastropods such as sea slugs. To eat its crusty prey, the rainbow wrasse has evolved sharp teeth and a protractile jaw.

A sequential hermaphrodite, the rainbow wrasse changes in color and size over its lifespan. These fish may be born either male or female, and in the primary phase they are colored brown with a white belly and a yellow-orange band running down either side of the body. Secondary-phase females reach a length of up to about seven inches, or they may change into secondary-phase males, and increase in size up to about 10 inches. Secondary-phase males are much more colorful—green or blue with a bright orange zig-zag stripe along either side.

The population of the Mediterranean rainbow wrasse is stable and not threatened. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.